2004

Q-Pod: Deployable and Scalable End System Support for Enabling QoS in Legacy Enterprise Systems

Asad Khan Awan

Kihong Park
Purdue University, park@cs.purdue.edu

Report Number:
04-005

Q-POD: DEPLOYABLE AND SCALABLE END SYSTEM SUPPORT FOR ENABLING QoS IN LEGACY ENTERPRISE SYSTEMS

Asad Khan Awan
Kihong Park

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #04-005
January 2004

Q-Pod: Deployable and Scalable End System Support for Enabling QoS in Legacy Enterprise Systems *

Asad Khan Awan † Kihong Park ‡
Network Systems Lab
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{awan,park}@cs.purdue.edu

CSD-TR 04-005
January 22, 2004

Abstract

Despite a growing need for a QoS-aware Internet, inter-domain policy barriers, legacy compatible deployment, and scalable integration stand in the way of realizing a workable solution. In this paper, we focus on the QoS requirements of enterprise systems and advance an incrementally deployable "turnkey" solution that achieves scalable QoS-to-the-desktop under variable network conditions. Our contribution is two-fold. First, we introduce an end system QoS support called Q-Pod that endows QoS capabilities to legacy applications running over legacy operating systems. At the heart of Q-Pod are its platform independent performance features that enable transparent user-directed QoS support with small overhead. The efficacy of Q-Pod's design features—implemented for Linux and Windows—is demonstrated using comprehensive performance measurements under high duress and varying workload conditions. Second, we show how Q-Pod integrates with QoS mechanisms exported by legacy routers to facilitate agile, scalable end-to-end QoS spanning user, application, end system, and network core. Q-Pod transforms a legacy enterprise system into a QoS-enriched counterpart without over-provisioning, utilizing existing network resources in concert with user requirements. We benchmark end-to-end QoS integration performance using representative enterprise applications—VoIP, real-time multimedia streaming CDN, and grid computing—over a 9-router testbed comprised of 7200 series Cisco routers.

*Supported.

†A. K. A.: Contact author; tel.: (765) 494-7811, fax.: (765) 494-0739.

‡K. P.: Additionally supported by NSF grants ANI-9714707, ESS-9806741, ANI-0082861 (ITR), and grants from AFRL F30602-01-2-0539 (DARPA ATO FTN), Santa Fe Institute, and Xerox.

1 Introduction

Large enterprises such as commercial, government, military, educational and research organizations have readily employed large scale intra-networks, and are increasingly relying on network technologies to conduct business at reduced costs, efficient and larger scales, and for providing value added services. Examples of sophisticated applications in this context are easy to enumerate, e.g., VoIP for lowering the cost of intra-enterprise telephony, grid computing for achieving faster computation capabilities, graded multimedia content distribution to home users for higher revenues, and broadcasting instructional multimedia to students on campus networks for out-of-classroom lectures. Such applications, each with specific quality of service (QoS) requirements, contend for shared, best-effort IP network resources. This necessitates provision for end-to-end QoS across the enterprise.

The key technical challenge in realizing a QoS-enriched enterprise, stems from the unavailability of an agile and efficient integration spanning the user, applications, end systems, routers in the network core, and administrators or service provider. This requires transparent, user-directed QoS endowment to legacy applications running on legacy operating systems, in concert with a network-wide QoS architecture including QoS mechanisms exported by commodity routers, and the essential service control and feedback. Due to the lack of a feasible solution, currently, over-provisioning is employed to allow smooth execution of legacy distributed applications, with heterogeneous QoS requirements, over best-effort IP. This solution is not scalable—neither in terms of cost-effectiveness nor QoS assurance. The latter statement has far reaching consequences on the utility of over provisioning—given a diverse user base with heterogeneous QoS requirements, an ever increasing number of distributed applications and the ubiquitous self-similar bursty nature of network traffic [24], unreliability and uncertainty in fulfilling heterogeneous QoS requirements is imminent in best-effort-only networks. Such a level of service for QoS sensitive applications is often unacceptable.

In this paper we present Q-Pod, which transparently and scalably transforms legacy enterprise infrastructure into a QoS-enriched system, by providing a deployable, end system based solution. Our first contribution is to address the ability to endow QoS capability, transparently, to legacy applications on legacy operating systems, incurring a low overhead. We present the platform independent, modular design of Q-Pod where our key focus lies in the high performance functional mechanisms and algorithms that comprise it. Q-Pod employs Q-Driver, a kernel level loadable (and hence easily deployable, requiring no kernel recompilation) network driver, and a user-space Q-Manager to achieve its objective functionality. We have devised mechanisms to dynamically map network sessions to their owning applications in order to provide fine-grained control, over these sessions, based on user supplied QoS requirements. We introduce a novel system call-free user-kernel communication mode (between Q-Manager and Q-Driver) using shared memory with lock-free consistency control under concurrent access. In addition, our data structure allows fast per-session information look-up allowing fast processing of network packets in the Q-Driver. It also allows maintaining low-level traffic measurements. We also describe our implementation on two popular operating systems—Windows XP and Linux. The high performance of our algorithms and the low overhead of Q-Pod mechanisms is shown using comprehensive performance evaluation of Q-Pod's prototype implementations. Our results include low level overhead measurements and evaluation under varying workloads and high duress, which show Q-Pod's scalability.

Our second contribution is to show how Q-Pod facilitates scalable end-to-end QoS, integrating the user, applications, end system, QoS mechanisms at legacy routers, and the service provider, on a "turn-key" basis. End systems exist on the edge of the network and Q-Pod installed on these systems provides *edge-control*. Q-Pod exports the ability to perform QoS architecture specific scalable per-application and per-session admission control on the basis of QoS requirements. Mechanisms implemented in Q-Driver allow QoS enforcement on the network sessions of applications, which may entail QoS mechanisms such as forward error correction, or QoS control messaging in form of IP type of service (TOS) label marking in DiffServ [10] environments or RSVP signaling in IntServ [34] environments. In the paper we describe Q-Pod's functional features in the

context of its edge-control responsibilities. We demonstrate Q-Pod supported network QoS, using benchmark experiments on a 9, CISCO 7200 series, router network testbed. The enterprise level, QoS sensitive applications that we consider in our benchmark experiments are VoIP, MPI based grid computing, and H.323 based real-time multimedia content distribution. We use Assured Forwarding [23] per-hop behavior supported in CISCO IOS 12.2 to provide differentiated QoS, utilizing which Q-Pod endows QoS capability to the legacy applications by TOS label marking.

1.1 Related Work

Q-Pod provides scalable and deployable end system QoS support for legacy applications running on legacy operating systems. The novelty of this system lies in the transparent QoS support for legacy applications running on legacy operating systems, and the algorithms and functional features aimed at achieving scalable performance through small system overhead. Q-Pod presents an efficient integration spanning the user and service provider, application, end system and network core.

Scalable network QoS provisioning: Architecting networks capable of providing scalable, efficient, and fair services to users with diverse QoS requirements is the focus of several research initiatives. The traditional approach uses resource reservation and admission control to provide both *guarantees* and *graded* services to application traffic flows. Analytical tools for computing and provisioning QoS guarantees [17, 18, 29, 30] rely on over-provisioning coupled with traffic shaping/policing to preserve well-behavedness properties across switches that implement a form of generalized processor sharing packet scheduling. The self-similar nature of network traffic [24] limit the shapability of input traffic while reserving bandwidth that is significantly smaller than the peak transmission rate. The overhead associated with administering resource reservation and admission control which require per-flow state at routers impedes scalability.

Recently, efforts have been directed at designing network architectures with the aim of delivering QoS-sensitive services by introducing weaker forms of protection or assurance to achieve scalability [12, 13, 19, 25, 28]. The differentiated services framework [10, 13, 27] has advanced a set of building blocks comprised of per-hop and access point behaviors with the aim of facilitating scalable services through aggregate-flow-resource control inside the network and per-flow traffic control at the edge.

End system QoS support: Most end system QoS support efforts have been directed towards reservation or fair allocation of resources such as CPU and memory [11, 22, 21, 26, 33]. Such work is mainly in the context of specific services, e.g., Web servers, multimedia servers and soft real-time applications. Features and limitations of several QoS architectures, specially in regard to QoS support for distributed multimedia systems, have been surveyed in [8]. In contrast, Q-Pod provides edge-control, by transparent QoS policy enforcement on the network packets of applications, which integrates into the network QoS infrastructure to harness end-to-end QoS. The scope of Q-Pod is not restricted to specific applications or network QoS infrastructures. QoS in CPU or memory allocation is out of our scope.

Network QoS initiatives such as IETF's Differentiated Services (DS) using IP type of service (TOS) field marking concern how QoS for network flows can be achieved using edge-control fulfilled by edge-routers at DS boundaries. However, the lack of scalable support for admission control and QoS policy enforcement for legacy application's network sessions at the edge of the network has been a key factor hindering QoS deployment. Q-Pod aims to fill this void.

Both Windows XP and Linux provide end system QoS support for applications. Windows XP provides APIs [15] using which applications can achieve services like packet marking, metering, policing etc. This restricts their utility to QoS-aware applications developed using these APIs. On Linux the Traffic Control [7] subsystem provides QoS facilities (e.g., queuing disciplines), amongst other network QoS support in the Linux kernel [32]. Command-line utilities are provided which can be used by an administrator to achieve transparent QoS by specifying port numbers or process identification. Port number information for legacy applications (e.g.,

peer-to-peer and client applications) may not be readily available except at run-time. Furthermore, the lack of an integrated edge-control component specially for admission control and per-session traffic measurements reduces the usability of these facilities in practical architectures.

Transparent legacy application support: IBM and Cisco, together, developed an architecture to provide transparent support for legacy applications running on the IBM S/390 Parallel Enterprise Server [2]. This system provides: (1) transparent QoS support for both QoS aware and non-QoS aware applications and (2) per-connection management and measurements. Q-Pod shares these two properties with this work. However, the IBM/Cisco system does not support dynamic session discovery and mapping, relying on static configuration files to detect sessions, which is a severe restriction in general enterprise and Internet environments. The design of IBM/Cisco's system where well-known port numbers may be available limits its application to legacy client platforms where port numbers are negotiated at run-time and can not be assumed given. Q-Pod dynamically discovers network sessions of legacy applications when they are initiated at run-time.

Another limitation of the IBM/Cisco system for S/390 servers is that it is tied with the proprietary IBM server and operating system (OS/390). The QoS support is built into the operating system's kernel and is compatible only with Cisco routed networks. Q-Pod, including its dynamically loadable Q-Driver, has been implemented for legacy operating systems—Linux and Windows XP, and is extensible to be integrated into different network QoS infrastructures.

2 System Architecture and Design Features

Q-Pod adheres with the end-to-end paradigm of IP networks, shifting management and measurements to the end systems, to achieve scalable yet fine-grained control over user's network flows. The core functionality of Q-Pod is to enable the user to specify QoS requirements for an application and transparently control the QoS treatment, received by each network session, in concert with an enterprise wide QoS infrastructure. The key principles behind the design of Q-Pod architecture are:

1. *Transparent and incremental deployability*: This is the key principle behind Q-Pod's architecture and encompasses:
   (a) Transparent support for legacy applications.
   (b) Implementable on legacy operating systems without requiring any modification to existing subsystems or kernel re-compilation.
   (c) "Turn-key" QoS-to-the-desktop support on existing networks, capitalizing on QoS mechanisms exported by legacy routers.
   (d) Modularity and extensibility to support custom QoS infrastructures, and implementation of transparent QoS mechanisms on end systems (e.g., forward error correction).
2. *Scalability and low footprint*: Q-Pod being a end system software which contends with user applications for end system resources, employs mechanisms and novel algorithms designed to exert a minimal footprint and scalable support for several processes and heavy network workloads.
3. *Platform independent design*: Q-Pod's modules and their interface specification, mechanisms, and algorithms are designed to be platform independent, demarcating Q-Pod's functionality from its reliance on operating system architecture. The platform specific procedures, required to transparently integrate with operating system's subsystems, are clearly abstracted. This is possible because, although the underlying architecture of modern operating systems differ, the functionality exported by them—in context of Q-Pod—is similar. To qualify this statement we enumerate the underlying architectural features that Q-Pod assumes:

(a) Layered kernel architecture with independent subsystems, making it suitable for implementation on both kernels with layered architecture (e.g., Windows XP) and monolithic kernels—though, not utilizing their, less restrictive, monolithic nature (e.g., Linux).

(b) Support for loadable kernel network modules or drivers.

(c) Support for "hooks" into the network data stream for packet interception.

(d) Kernel-space protocol stack (systems with user-space protocol stacks can be trivially supported).

(e) Support for user-kernel shared memory—a feature ubiquitously supported on platforms which implement virtual memory.

In the rest of this section we discuss the architecture of our system in terms of functionality, design, performance features and algorithms, and interactions of the modules that comprise it. We also provide the design choices and the motivation for the different functional components of our system.

2.1 End System Based Network QoS Support

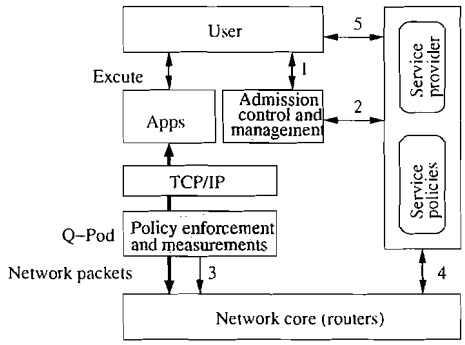

Figure 1: Big picture: Q-Pod based QoS architecture.

End systems exist on the edge of a network and Q-Pod installed on these systems provides *edge-control*, which integrates into the network QoS architecture. (Refer to Figure 1). (1) The users of these end system specify the QoS requirements, in the form of *target end-to-end QoS parameters*, for the network applications they run. Edge-control exported by Q-Pod consists of access control, QoS policy enforcement and per-flow measurements. Access control consists of admitting applications with QoS requirements based on per-application admission control rules, dynamically discovering the sessions initiated by these applications and thence mapping each session to a QoS policy based on QoS infrastructure specific per-session admission control rules. This QoS policy when enforced on the network sessions of the user's application will achieve specified target QoS, under the limitations of the QoS infrastructure. A service level agreement (SLA) or contract, may need to be negotiated with user at the time of admitting applications and their QoS requirements, or a priori. (2) The per-application and per-session admission control rules may be set or dynamically updated by the network service provider and hence, Q-Pod integrates into service provider control. Similarly, they may be evaluated in a distributed fashion via collaboration between Q-Pod's. Q-Pod provides per-flow measurements which may be required in this context. Fine-grained measurements are of key utility for several reasons, such as pricing, planning, verifying SLA conformance, and for monitoring and auditing network usage.

In a QoS provisioning architecture sufficient and efficient QoS mechanisms are required such that user QoS requirements can be fulfilled. QoS mechanisms such as packet scheduling, or forward error correction can be implemented in Q-Pod to provide transparent QoS to network sessions of legacy applications. More importantly,

Q-Pod integrates with standard QoS mechanisms exported by legacy routers to facilitate agile and scalable end-to-end QoS. (3) Q-Pod communicates QoS policies reflecting user requirements, via some form of QoS control messaging, to the routers implementing the QoS mechanisms. (4) These legacy routers can configured by the service provider. For example, in a *DiffServ* environment the routers may provide differentiated treatment of packets based on the labels inscribed in the TOS field of the IP header, and end-to-end QoS is determined by the global effect of the *per-hop control*. The label values (QoS policy) are ascertained (admission control) and set in the IP header of packets (policy enforcement), on a per-flow basis by Q-Pod, while the routers provides *per-hop behavior* for aggregate flows to achieve scalable end-to-end QoS. In a similar way Q-Pod can also provide RSVP control signaling with routers, on behalf of the network sessions of the legacy application to provide *guaranteed service* in an *IntServ* environment.

Finally, (5) social interface between the users and the service providers, e.g., satisfaction, or monetary payment serves as a social control and feedback for the QoS architecture.

Based on this high level description of Q-Pod enabled scalable QoS architecture we move on to the technical description of the construction of Q-Pod modules and their interaction on an end system. We focus on the high performance mechanisms and algorithms required to transparently support legacy applications on legacy operating systems.

2.2 Transparent and Seamlessly Deployable QoS Support

Based on the edge-control tasks summarized earlier, we have partitioned the responsibilities of Q-Pod into three modules, namely Q-Interface, Q-Manager and Q-Driver. This affords platform independence and performance due to careful delegation of tasks to each module, and extensibility due to modular design. The *Q-Interface* acts as an interface between the user and Q-Pod by allowing the user to easily specify the application and its QoS requirements and interact with our system. The *Q-Manager* is the central control of the Q-Pod on an end system. It is involved with receiving the application execution request and its QoS requirements from the Q-Interface, and based on an application admission control scheme accepting the user's request, after producing an appropriate service level agreement and a contract, where by a user might be required to pay a cost for receiving the service, which the user accepts. Furthermore, when the application is executing it might engage in several network sessions, appropriate per-session admission control, in terms of mapping the session to a QoS policy to achieve the target QoS, is performed by the Q-Manager. Finally, logging, of fine grained per-session traffic measurements to files, is performed by the Q-Manager. Unlike the Q-Interface the Q-Manager is executed and controlled by the administrator of an end system. Transparently and dynamically discovering the network sessions of an application and QoS policy enforcement (e.g., IP packet TOS marking) on these sessions requires access to the network subsystem in the kernel of an operating system. Thus, we introduced the *Q-Driver* as a kernel level loadable network driver which intercepts all UDP and TCP packets[1] being received or transmitted by the end system. The Q-Manager and the Q-Driver are tightly coupled with respect to the communication between them. The communication between the Q-Manager and Q-Driver—including per-session QoS policies provided by the Q-Manager and network traffic measurements provided by the Q-Driver—is done via shared memory using our lock-free algorithm. This has the dual goal of achieving high performance and platform independence (Section 2.3.2). We also allow for an event based control notification between the Q-Driver and the Q-Manager used for dynamic discovery of new sessions (Section 2.3.1). To achieve seamless deployability on legacy operating systems the Q-Driver is designed as a kernel driver (loadable kernel module on Linux and network driver on Windows XP) which can be easily installed at run-time, while Q-Manager and Q-Interface are user space applications. As the Q-Manager is a user-space program it can be easily extended to add intelligence such as admission control

[1] It must be noted that the design of Q-Pod is independent of the protocols used by applications, however, we have restricted the scope of our current implementation to the TCP/IP protocol.

schemes and accounting, while the Q-Driver provides efficient and low footprint packet processing based on policies provided to it by the Q-Manager. Figure 2 shows a schematic view of a Q-Pod enabled end system,

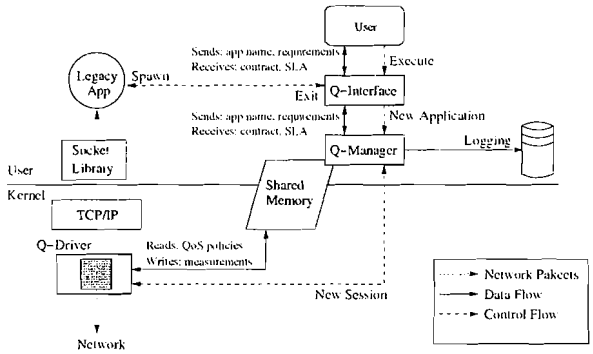

Figure 2: Interactions—data flow and control flow—in a Q-Pod enabled end system.

showing the conceptual placement of the Q-Pod modules and their interaction with the user, legacy application and the network subsystem. Note the placement of the Q-Driver below the TCP/IP layer, indicating that it is able intercept and affect all, incoming and outgoing IP packets via the `intercept_outgoing_packets()` (IOP) and the
`intercept_received_packets()` (IRP) functions represented by the darkly shaded box in the Q-Driver. The responsibilities entrusted on the Q-Driver have dictated our choice for the conceptual placement of the Q-Driver below the IP layer, specifically the diversity of the QoS policy enforcement and QoS mechanisms that can be implemented depends on this position. For example, marking TOS field of the IP header and choosing QoS policies based on destination network address can easily be achieved once the IP stack of the OS is done with the packet.

2.2.1 Transparent Packet Interception and Processing

Given that the interception functions in the Q-Driver acquire access to the IP packets being transmitted or received by an end system, three mechanisms need to be implemented—(1) support for dynamic network session discovery, (2) QoS mechanisms to enforce QoS policies on the network packets and (3) per-flow traffic measurements. The architecture of the interception functions (IRP and IOP) to implement these mechanisms is described here.

The IRP and IOP functions keep track of all the sessions that an end system engages in. A session is described by the quintuple {local IP address, local port number, remote IP address, remote port number, protocol type}. Keeping track of all the sessions is needed to be able to dynamically recognize a new session. For performance and management considerations, which will be duly explained, we categorize the packets as belonging to *filtered*, *managed* or *unmanaged* sessions. Filtered sessions belong to a small static list of of sessions, with known port numbers and protocol types, which will never require specialized QoS support. Managed sessions are those which belong to applications with QoS requirements, i.e., for which per-session QoS policies need to be enforced. Finally, all remaining sessions are unmanaged sessions. A default QoS policy, e.g., best-effort, can be associated with all filtered and unmanaged sessions. An entry for each managed session is kept in the *Session Table* in the shared memory, where all measurements and the QoS policy for the session is kept.

Figure 3 illustrates the architecture of the interception functions. As can be seen from the figure, an intercepted packet is first checked if it is a the filtered session, hence filtered sessions are processed without much overhead. The `check_managed()` function retrieves the index of the managed session information in the Session

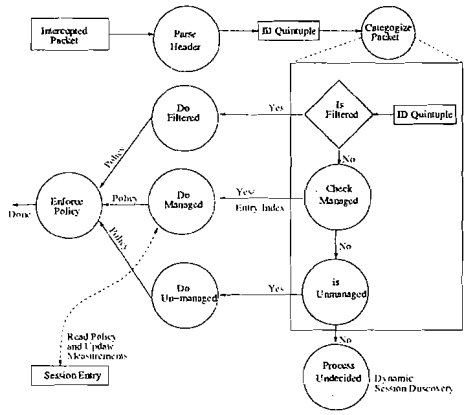

Figure 3: The processing of packets in the interception functions. The packets headers are parsed to get the identification information about the session it belongs to. The id is used to categorize the packets and perform processing based on the category.

Table, if an entry for this session exists. The managed session classification (entry lookup) is done using a hashing scheme described in Section 2.3.3. The managed session's QoS policy is retrieved and the measurements are updated in its entry. Information for identifying unmanaged sessions is kept in the *Unmanaged Session List*. If a session doesn't belong to either of these categories then it must be a new session. Such sessions are called undecided sessions because at this point it is not known if they are to be treated as managed or unmanaged sessions. A notification of the newly discovered sessions is sent to the Q-Manager, which associates a session with an application, and if the application has a QoS requirement it will assign the session a QoS policy, based on the per-session admission control scheme, and create an entry for the session in the Session Table. If the session belongs to an application which does not have specified QoS requirements, an entry for this session will be created in the unmanaged table. Subsequently, all packets will be treated, by the Q-Driver, based on this assignment. This mechanism is called Dynamic Session Discovery (DSD). The complete details and the performance oriented design of DSD of is discussed in Section 2.3.1.

2.3 Performance Oriented Design Features

2.3.1 Dynamic Session Discovery

The notification of a new session to the Q-Manager, mapping the session to an application and if the application has QoS requirements mapping the session to a QoS policy based on per-session admission control and creating an entry for the session in the Session Table or Unmanaged Session List needs to be performed in a timely fashion. This is because while these tasks are being performed none of the packets belonging to the session can be processed and hence will be dropped. It is necessary to drop the packets belonging to these undecided sessions because of semantic correctness of our system—all packets being received or transmitted by an end system conform to the QoS contract accepted by the user, if the QoS policy for the session is not known, it will not be processed. The sensitivity of QoS architectures and certain QoS policies (e.g., "drop all packets for the session"), requires establishing such stringent semantics.

The design details of DSD which make it efficient are as follows. A dedicated thread of the Q-Manager—Session Discover and Management (SDM) thread waits for notifications, of newly discovered sessions, from either of the two interception functions of the Q-Driver. The notification is sent using platform provided kernel-user events. Such events are attractive for use because the usually affect the scheduling such that a thread waiting

for kernel event is usually given priority for running, hence expediting the event processing. Such events usually do not allow transfer of data over the user-kernel boundary, making them efficient. However, in our case we need to send information (the id quintuple) of the newly discovered session with the notification. For this we have employed lock-free single-reader, single-writer Event Data Queue (EDQ) in the shared memory, where information of each discovered session is enqueued when the notification is sent. Furthermore, we employ event batching such that if the SDM is already awake and is processing events, new events are not triggered and only the session information is enqueued in the EDQ. This is beneficial in high load cases (self-similarity of session arrivals [20]) where several new sessions are discovered almost simultaneously. The batching of events and use of EDQ for data transfer makes our notification scheme efficient.

When a notification is received by the SDM it needs to map the session to an application. On monolithic kernel based OSes such a task can also be performed by the Q-Driver, however, on layered or micro kernel based OSes such a task can not be performed by a low layer network driver. To keep the system platform independent we opted that session-to-application mapping be done on in the user-space SDM thread. Mapping a session to an application requires scanning the open sockets and their owning applications and matching the session to the socket using port numbers and IP addresses. This is intrinsically a time consuming process. Our performance results show that this processing takes up bulk of the approximately 4 ms processing cost of DSD (Section 4.4).

To ameliorate the delay (and dropping of packets arriving in this interval) overhead incurred due to this processing we save the first packet of an undecided session and forward it once the packet session is evaluated to be managed or unmanaged. This has important implications. For TCP sessions, the first packet saved is the `SYN` packet, if this packet is dropped the retransmission takes in the order of few seconds. Saving this packet and forwarding it within few milliseconds (at the end of DSD) avoids the huge retransmission overhead. For UDP sessions, usually the first packet contains setup information (as in multimedia applications), hence saving this packet and forwarding it can avoid overheads. Furthermore, it results in at least one packet being forwarded for even very short UDP sessions. It should be noted that the packets intercepted for a session whose DSD is in progress will be dropped. Nonetheless, this does not violate UDP's semantics which dictate that packet transmission is unreliable.

2.3.2 User-Kernel Communication Using Lock-Free Shared Data Structures

The high frequency of data communication—DSD data, per-managed session data (QoS policies and measurements), per-unmanaged session data—between the Q-Driver and the Q-Manager threads dictates the need to use a low overhead communication mechanism. The use of lock-free[2] shared memory was opted because it affords higher performance, easier programmability—by avoiding use of complex kernel level locks, and platform independence. Performance benefits stem from avoiding system calls (for either I/O or acquiring kernel locks) and blocking. Thus, eliminating overheads due to data copying, reduced concurrency, context switching, page faults, cost of executing extra (system call) code, and kernel level locking (which affects the whole system, e.g., because of disabling `softirqs` as in Linux). We claim platform independence of our scheme because it simply defines access to shared memory without using operating system constructs. However, setup of shared memory between the user and kernel space requires assumptions about the underlying platform. They are: (1) shared memory pages should be set as non-pageable, (2) a process-context-independent address of the shared memory should be available for use by the Q-Driver, (3) direct access to the shared memory should be allowed, i.e., no buffering should be introduced and (4) the pages can be unlocked and unmapped when Q-Pod stops. Most of modern operating systems fulfill these assumptions.

In the context of this section user-kernel communication entails the Q-Manager updating new data to be used by the Q-Driver, and reading data being written by the Q-Driver. Q-Driver does not "pull" data from the

[2]We avoid the use of any operating system based locks, such as, spin locks, IRQ level based locks and semaphores, or system calls.

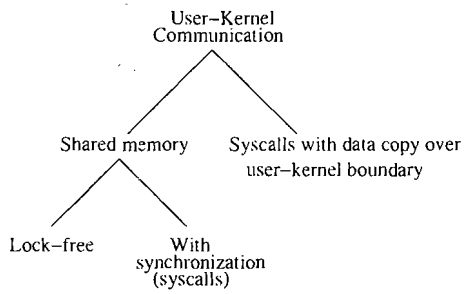

Figure 4: A taxonomy of user kernel data communication techniques.

Q-Manager (except in the case of DSD where a kernel-user event is required), rather only passively reads data provided by the Q-Manager each time it processes packets. Figure 4, gives a simplified taxonomy of user-kernel data communication methods which can be used in this scenario. Most application and driver implementations, for legacy operating systems, either use system calls requiring data copy over user-kernel boundary or shared memory synchronized using system calls often invoking locks. When using system calls the user-space programs and kernel-space drivers each keep a private copy of the data structures. The user-space programs use system calls to read or write data to the kernel level drivers copying data over the user-kernel boundary keeping the two copies consistent. The other alternative is using shared memory. With the use of shared memory the first issue that arises is data consistency control under concurrent access. Note that in this scenario there is concurrent access to the shared memory by user-space threads and kernel-space driver. This is of concern because user-space threads can always be arbitrarily preempted by the kernel level code (and hence holding a kernel level lock in user-space will cause a deadlock). This implies exclusive access of user-space programs to the shared memory can not be guaranteed. The only option is to use a system call to trap to the kernel, where a kernel level lock which makes sure that the driver will not execute can be obtained. While holding the lock synchronization can be performed. For example, in a single call both reading from and writing to the driver's private data structures may be performed, avoiding extra system calls and costly copies over the user-kernel boundary, and in many instances avoiding copying data which is already consistent—take the case where measurements for an inactive network session need not be copied. Nevertheless, this synchronization is costly because of the extra memory copying, use of kernel lock which prevents the driver (and hence the network stack) from executing and most importantly the use of a system call. In fact our performance results, in Section 4.2 indicate that the major cost is attributable to the use of system call. Our lock-free and system call-free algorithm provides a mechanism which allows user-kernel data communication avoiding system calls, locks and data copying.

2.3.3 Lock-free Session Table

We provide a lock-free hash-indexed shared data structure which allows efficient contention-free entry creation, deletion, search, read, and write operations under concurrent access, by Q-Manager threads and the Q-Driver interception functions.

Definitions: The Session Table is a static array of entries for managed sessions. An entry, alloted per-session, is the set

`{vflag, dflag, {ID}, {pol}, {meas}}`. The `vflag` is a boolean indicating the validity of the entry. The `dflag` is a boolean which when set to false indicates that the entry deletion is underway causing subsequent concurrent access to the entry to fail. The `ID` set is the quintuple giving remote and local IP and port, and the protocol (UDP or TCP). The `pol` set contains elements representing the QoS policy for this session. The `meas` set contains several elements containing traffic measurements for this session. Table 1 shows the primitives

for operations over Session Table. If all these commands are allowed to execute concurrently we will have contention resulting in possible inconsistency as shown in Table 2. (Discussion of hashing scheme including explanation of `GIVE_UP` and `tries` is deferred.)

Table 1: Primitives for Session Table entries.

`get_free_ent(ID)`	`while(tries<GIVE_UP)`
	`  e:=hash(ID,tries);`
	`  if(e.vflag=false)`
	`  return e;`
	`  tries++;`
	`return GIVE_UP;`

`get_ent(ID)`
(Search entry using ID as a key)
	`while(tries<GIVE_UP)`
	`  e:=hash(ID,tries);`
	`  if(e.vflag && e.dflag &&`
	`  ID=e.ID) return e;`
	`  tries++;`
	`return GIVE_UP;`

`create_ent(ID,pol)`	`e:=get_free_ent();`
	`write(e.ID, e.pol);`
	`clear(e.meas);`
	`e.dflag:=true; e.vflag:=true;`

`delete_ent(ID)`	`e:=get_ent(ID);`
	`e.dflag:=false;`
	`clear(e.ID); e.vflag:=false;`

`read_ent(ID,X)` `/*X: meas or pol*/`	`e:=get_ent(ID);` `return(e.X);`
`rw_ent(ID,d,X)` `/*d: new data*/`	`e:=get_ent(ID);` `e.X:=op(e.X,d);/*operation*/`

Precedence relationships: The following precedence relationship constraints are imposed by most of modern commodity operating systems.

(C1) User space programs can preempt each other arbitrarily.

(C2) Kernel code can preempt user code arbitrarily.

(C3) User space programs can not preempt kernel code.

(C4) On some platforms (e.g., Windows XP) IRP can preempt IOP as it is invoked at a higher IRQ level.

(C5) At a given time more than one IOP or more than one IRP can not be executing (on single processor machines).

Rules and their implementation: We provide the following rules and their efficient implementations—capitalizing on the precedence relationship constraints given above and use of lock-free data structures, to maintain data consistency.

(R1) All entry creations be performed by a single user-space (SDM) thread.

(R2) Writing to ID fields is not allowed except at entry creation.

(R3) All entry deletions be performed by a single user-space thread.

(R4) Each set should have a single-writer, i.e., non-concurrent writers. Using the precedence relationship constraint C3 and C5 all writes executed in the kernel space follow this rule. An exception to be noted is that IRP and IOP both write to `meas`, and per constraint C4, IRP can preempt IOP. This can be easily resolved using

Table 2: Adversaries in contention. "1" in a cell indicates that corresponding operations if run concurrently can cause inconsistency. (c=create, del=delete, r=read, w=write, X=meas or pol)

⊕	w-X	r-X	del	c
c	0	0	0	1
del	1	1	1	
r-X	1	0		
w-X	1			

Example: cell(w-X,w-X)=1
If two writers (e.g., IRP and IOP) concurrently write to the same set X (e.g., meas) data may become inconsistent.

careful design of data structures, e.g., breaking down the set `meas` into two sets: `r_meas` (writable by IRP only), and `o_meas` (writable by IOP only) resolves contention.

(R5) Entry deletion can not preempt read or write. Given this rule and the use of `dflag`, contention of delete and read/write can be eliminated. Given that deletion is done in user space and that precedence relationship constraint C3 holds, this rule is satisfied, without further ado, for reads/writes done in kernel space. A simple way to enforce this rule for user space read/write is to perform deletion, read, and write in the same thread. As deletion frequency is much less than read/write frequency this is a feasible technique.

(R6) *Serializability property*[3] should hold for concurrent reading and writing of a given set (such that all elements in the set are consistent). The challenge is to achieve this without using locks. Given that a set contains more than one element, the `read_ent(ID)` and `rw_ent(ID,X)` primitives as shown in Table 1 fail this requirement and need revision. We propose the use of a single-reader-single-writer circular queue to hold the data for each set. This data structure is lock-free because it keeps only two variables to access the queue (including calculation of length)—`top` and `tail`, with exclusive write access of `top` given to the writer and that of `tail` given to the reader. Table 3 shows the necessary revisions. Given that the Q-Manager updates the QoS policies, using `u_rw_ent()`, slower than the packet rate, i.e., the rate at which the kernel reads these values, the queue length remains small. However, the Q-Manager might not be able to keep pace with Q-Driver measurements and hence we use `k_rw_ent()` for updating entries in the kernel which caps the queue length at 2. This is possible because Q-Manager can not preempt Q-Driver interception functions (per constraint C3). For traffic measurements often a complete trace is required, rather than just the last measurement or cumulative measurements. Using `t_read_ent()` and `t_write_ent()` affords this for free when using single-reader-single-writer circular queue. This allows collecting measurement sequences at a high speed, by affording a writer inserting measurements in the queue each time a packet is intercepted, and a reader deleting from the queue in batches.

Correctness: Based on these rules and the provided implementations we rewrite Table 2 to give Table 4. This table shows the correctness of our approach in attaining lock-free consistency control under concurrent entry create, delete, search, read, and write operations.

Fast classification hashing scheme: Hashing is employed to allow for fast searching (and hence fast packet classification) of an entry corresponding to the ID key. It is important to note that not all sessions have an entry in Session Table (it is for managed sessions only), hence the search should be able to report that an entry was not found. We have used a simple hashing scheme which resolves collisions by probing—for each try a different hash calculation is performed to generate a different (entry) index. The i^{th} retry always uses the i^{th} hash calculation method. Retries are performed for `GIVE_UP` number of times. This implies that the `get_free_ent(ID)` function may not find a free entry, within these retries, resulting in the `create_ent(ID)` function not creating an entry for this session in the Session Table. The `get_ent(ID)` follows the same probing

[3]Meaning, concurrent execution of conflicting transactions, producing a result which is the same as if they were executed serially.

Table 3: Read and write primitives using a lock-free single-reader-single-writer circular queue.

queue operations	insert(q,d), d:=delete(q) l:=length(q) d:=peek_top(q) /*reads top without deleting*/ write_top(q,d) /*overwrites data in top*/

read_ent(ID,X)
/*does not delete
top and gets
latest value*/
	e:=get_ent(ID);
	while(length(e.X.q)>1)
	delete(e.X.q);
	return(peek_top(e.X.q));

k_rw_ent(ID,d,X)
/*allows kernel
to update
while capping
q length at 2*/
	e:=get_ent(ID);
	if(length(e.X.q)>1)
	d:=op(peek_top(e.X.q.d),d);
	write_top(d);
	else insert(e.X.q,d);

u_rw_ent(ID,d,X)
/*allows user
to update*/
	e:=get_ent(ID);
	d:=op(peek_top(e.X.q.d),d);
	insert(e.X.q,d);

t_read_ent(ID,X) /*trace data*/	e:=get_ent(ID) return(delete(e.X.q));
write_ent(ID,d,X) /*trace data*/	e:=get_ent(ID); insert(e.X.q,d);

sequence as in `get_free_ent(ID)`, until a valid entry (both flags are true) whose ID matches the given ID is found. Exhaustion of retries without finding a match implies that there is no entry for this session in Session Table, i.e., the session is not managed—either because it had no QoS requirements or because entry creation failed. The problem of entry creation failing even when there are free slots in the table can be resolved by adding a small array to the table where entries can be created sequentially, however this has a performance tradeoff. Increasing the size of the table, and hence decreasing the probability of the entry creation failure has the tradeoff of under-utilized shared memory. In our implementation we used hash functions of the form $hash(x,i) = (a_i x + b_i) mod p$, where p, a prime number, is the table size. a_i and b_i have to be chosen considering characteristics of the key x. For $p = 127$ and $GIVE\_UP = 4$ we are able to achieve 90% utilization. Finding out that an entry is not present has a very low overhead (compared to sequential search) as only four retries are used.

2.3.4 Other Data Structures in Shared Memory

Unmanaged Session List: An entry in the Unmanaged Session List contains only one variable: the local port number. This is sufficient to unambiguously identify packets from an unmanaged application. The deletion and creation of an unmanaged session entry is done in the same user space thread and thus does not require synchronization. The deletion of an entry (which occurs after the socket closes) involves clearing a single variable, hence access to the entry before the variable is cleared is legal, and access after the variable is cleared will result in a miss and thus faithfully invoke DSD.

Event Data Queue: EDQ is a single-reader-single-writer lock-free circular queue. The reader of EDQ is the SDM thread. The writer is the interception function. As we have two interception functions, two EDQs may

Table 4: Lock-free contention resolution. Rules (R) used to resolve contention are shown in cells.

⊕	w-X	r-X	del	c
c	0	0	0	0:R1
del	0:R5	0:R5	0:R3	
r-X	0:R6	0		
w-X	0:R4			

Example: cell (r-X,w-X)=0 utilizing implementation of R6 which allows concurrent read and write while maintaining the serialize property.

need to be used, to avoid synchronization between the two writers of the EDQ on platforms where the IRP can preempt the IOP.

3 Implementation

We have implemented Q-Pod on Linux 2.4.x with Netfilter and Windows XP. In our implementation the Q-Driver provides QoS enforcement on the packets in the form of IP type of service (TOS) field marking. Underneath its platform independent features, Q-Pod relies on the network and I/O subsystems, and driver architecture which are specific to each operating system. As mentioned earlier, most commodity operating system fulfill the assumptions about underlying platform features, made in Q-Pod design. In this section we describe the architecture of relevant Linux and Windows XP subsystems, and how we utilized them to achieve the functional goals of Q-Pod.

3.1 Linux Q-Pod Implementation

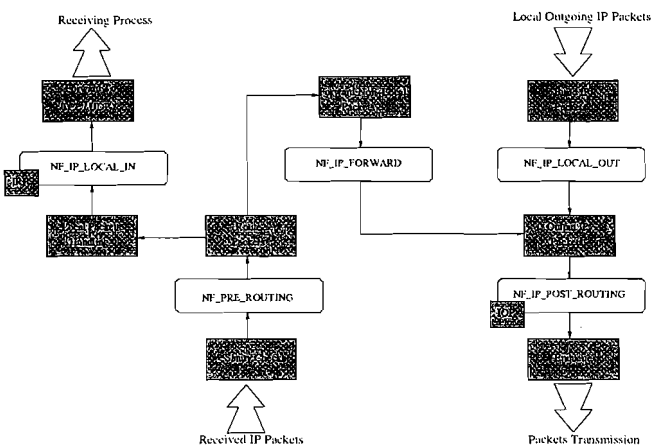

Figure 5: Packet handling by the Linux IPv4 stack, with Netfilter hooks.

The Q-Driver is developed as a Loadable Kernel Module (LKM) on Linux. Transparent packet interception is achieved by hooking our interception functions into the Linux IP protocol stack using Netfilter [4], which is available on most Linux distributions. The Linux IP stack is implemented as a sequence of function calls.

A packet traversing the IP stack is passed to the Netfilter's hooks in its path, which in turn calls the functions registered with that hook. Figure 5 shows the hooks of Netfilter in the IP stack and the hooks that are used for IRP and IOP functions. Netfilter passes intercepted packets encapsulated in the `sk_buff` structure to IRP and IOP. This gives us access to the headers which we can process as per the packet processing architecture given earlier. Recall that the conceptual placement of the Q-Driver is below the IP layer; although the chosen hooks are inside the IP stack their location allows us the same control as interception of packets below the IP layer. Developing a LKM which registers to Netfilter hooks makes Q-Pod easily deployable at run-time on any Linux 2.4+ system with Netfilter. We use a character device created for our driver to perform IOCTL with the driver. In the Linux kernel we are able affect the scheduling of the user space threads. We use this ability to develop our *own* kernel-user event mechanism which supports timeouts—the SDM thread sends a message to the Q-Driver using IOCTL, which gives us access to the thread's task pointer in the kernel. Using this pointer we put SDM to sleep with a timeout. If the Q-Driver needs to send a notification to the SDM thread, the thread is simply woken up by putting it into the ready queue, and the scheduler is invoked. If the timeout elapses the thread is woken up by the Linux kernel itself. This is an example where we have utilized platform specific features to achieve efficiency while maintaining the abstraction defined by Q-Pod design. To setup the shared memory we create a memory in the kernel and set it to non-pageable. The Q-Manager gets an address to access the shared memory by using `mmap()` call, which is handled by the Q-Driver character device. We utilize the `/proc` file system to map sockets to process ids, to be used in DSD by the SDM thread.

3.2 Windows XP Q-Pod Implementation

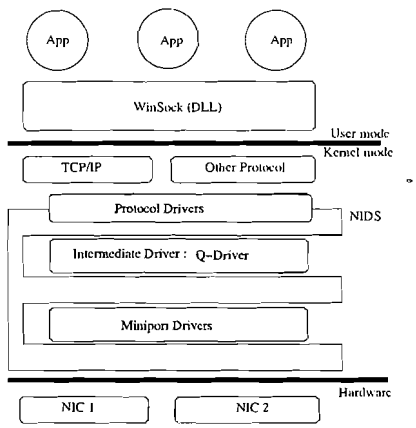

Figure 6: The NDIS in the Windows XP packet forwarding path. Note the relationship of the drivers at different layers within the Windows XP Network Architecture. The Q-Driver is implemented as an Intermediate NDIS driver.

Windows XP has a layered kernel architecture, as compared to the monolithic kernel of Linux. Using a layered approach has programmability benefits which trade off with performance. Low level network drivers for Windows follow a set of driver development specifications, and a set of interface specification using APIs given by NDIS (Network Driver Interface Specification). This makes the task of NDIS driver development arduous and complex. Nonetheless, the driver design, and the network subsystem itself is very modular. Due to the layered nature of Windows XP kernel it is possible to program drivers which fit into the packet forwarding path at different levels. We choose to intercept the packet as low in the path as possible, thus we developed Q-Driver as an NDIS Intermediate Filter driver [16], which lies below the TCP/IP layer. In Figure 6 we illustrate the location of NDIS in a simplified view of Windows XP network subsystem. We register our interception

functions with NDIS such that all packets leaving or entering the machine are forwarded through the Q-Driver. The intercepted packets are encapsulated as shown in Figure 7, thus the buffer lists need to be traversed to access the complete headers.

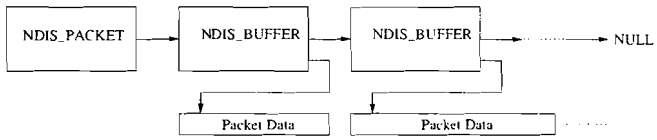

Figure 7: An NDIS packet.

I/O between the driver and user programs is done via the Windows XP I/O subsystem which provides a uniform interface. It has a layered architecture where requests are sent wrapped in I/O request packets. We use this to perform IOCTL with our driver and also to setup the shared memory. The user program can pass a memory address (and its size) which it wants to share. This is encapsulated as a Memory Descriptor List inside the request packet and sent to the Q-Driver. This memory is automatically made non-pageable and system context addressable. For kernel-user events and socket-to-process mapping (by the SDM), we use Windows provided APIs.

4 Performance Evaluation

In this section we evaluate the performance of Q-Pod. We demonstrate the efficacy of the key performance feature of Q-Pod—lock-free shared memory based user-kernel communication, and the low-footprint of its functional mechanisms—packet processing and session management.

We show the high performance and scalability of our lock-free shared memory based user-kernel communication by analyzing it in contrast with user-kernel communication based on shared memory with synchronization and using system calls.

We also present the low-level measurements of the overheads of Q-Pod on the network sessions of an end system. An evaluation of the impact of the overheads is provide using traffic level measurements under varying workloads and high duress, demonstrating the scalability and wide operating range of Q-Pod.

The performance evaluation of Q-Pod has been performed using our implementations on Linux and Windows XP. We present our results for Q-Pod's evaluation on Linux, and analyze differing results, due to platform dependent features, for Q-Pod's Window XP implementation.

4.1 Experimental Setup

The experiments described in the following sections were conducted using two x86-based machines, each with a Pentium 4 processor at 2 GHz and 1 GB RAM. The CPU has a 12 KB L1 instruction cache, a 8 KB L1 data cache and a 256 KB L2 cache. Both machines are equipped with 3COM PCI 100 Mbps Ethernet network interface cards (NIC). The two machines are connected via a 100 Mbps switch. The operating systems used are Linux 2.4.21 (with Netfilter) and Windows XP Professional. Low level time measurements were taken using the `rdtsc` x86 assembly command [14], which gives nanosecond granularity. For some microsecond granularity measurements `gettimeofday()` was used. The Q-Pod used for evaluation uses a lock-free Session Table (Section 2.3.3) which does not use the trace functionality of the single-reader-single-writer circular queue, thus a queue of max length equal to two is used.

4.2 User-Kernel Communication

To evaluate the performance of lock-free shared memory based user-kernel communication in contrast with other user-kernel communication methods we developed two other versions of Q-Pod using shared memory with synchronization using system call and read/write system calls for user-kernel communication. The system call is implemented by making handlers for standard operations on character devices, in the Q-Driver. Thus, `open()` using the Q-Driver character device name gives a file descriptor which can be used in standard system calls such as `ioctl()`, `read()`, `write()` to communicate with the Q-Driver. Q-Pod using shared memory with synchronization was developed such that the Q-Driver handler invoked by the system call locks the kernel data structures and synchronizes the pertinent part of the shared memory before it is accessed. Q-Pod using read/write system calls were implemented such that each read/write operation requires two system calls—one to inform the Q-Driver of which part of the shared memory is going to be accessed and second to actually perform the read or write. This reduces the size of data passed over user-kernel boundary. We also chose `Iptables-1.2.9` [4] (which uses Netfilter for intercepting and processing packets) for comparison. Iptables also uses two system calls for each of reading and writing. The first gets the size of the kernel data structure and the second reads or writes the complete data structure. These cover the all the user-kernel communication modes shown in Figure 4.

Q-Pod with Lock-Free shm	Q-Pod with synch. shm	Q-Pod with read/write

`start_r:=rdtsc`
`d:=read_ent(ID,X)`
`end_r:=rdtsc`
	`start_r:=rdtsc`
	`syscall(ID)`
	`d:=r_shm(ID,X)`
	`end_r:=rdtsc`
		`start_r:=rdtsc`
		`write(ID)`
		`read(d)`
		`end_r:=rdtsc`

`start_w:=rdtsc`
`write_ent(ID,d,X)`
`end_w:=rdtsc`
	`start_w:=rdtsc`
	`w_shm(ID,d,X)`
	`syscall(ID)`
	`end_w:=rdtsc`
		`start_w:=rdtsc`
		`write(ID)`
		`write(d)`
		`end_w:=rdtsc`

Figure 8: Calculating the time consumed for user-kernel communication.

Table 5: CPU time consumed for user-kernel communication using different mechanisms.

Mechanism	Cost (μs)
Lock-free shared memory (read)	0.41
Lock-free shared memory (write)	0.41
Shared memory w/ synchronization (read)	2.23
Shared memory w/ synchronization (write)	2.23
Read	4.70
Write	4.79
Iptables - read	21.0
Iptables - write	35.0

Measurements were performed using `rdtsc` bounding only the *communication operation* (Figure 8). When the system is mostly idle this time can be assumed to be the CPU time consumed by these operations, specially if average is taken over several measurements. The average over 100,000 measurements of CPU times consumed by each of these mechanisms is presented in Table 5. The measurements show that user-kernel communication using lock-free shared memory performs around six times better than its counterpart using system call for syn-

chronization. Most of the overhead in using this mode arises from the use of the system call, as the processing time within the system call handler was only 0.5 μs. The cost of reading and writing in the shared memory cases is the same because copying from local variables to shared memory or vice versa takes approximately the same time. The cost of reading and writing using system is call is almost double that of the cost of using shared memory with synchronization because it uses two system calls per operation. The results of Iptables are much greater than for Q-Pod because of different data structures that it uses and because it copies the whole data structure over the user-kernel boundary on each access.

System calls serve as a point for the invocation of context switching. In our case this implies, an increase in probability of context switching while the user-kernel communication operation is executing, affecting the time taken for the operation to complete. To analyze this behavior we measured the time for completion of the operation similar to our previous experiment, with the addition of running varying number of simultaneously launched UDP CBR packet generators, generating 64 bytes—including all headers—packets at an aggregate rate of 50 kpps (kilo packets per second). This traffic generator does not use sleep, instead does a busy wait for the inter-packet generation time. This is required to achieve CBR traffic at such high packet rates (the granularity of sleep calls varies up to 10 ms or a jiffie, which is the kernel timer granularity, and they do not provide guarantees). Thus if two such generators are run the aggregate rate is the same as that for one—at any time only one of them is running and each faithfully produces CBR traffic using only inter-packet generation time. The results for user-kernel communication using lock-free and synchronization based shared memory, and for Iptables read and write operations are presented in Figure 9. The time for completion of user-kernel communication using

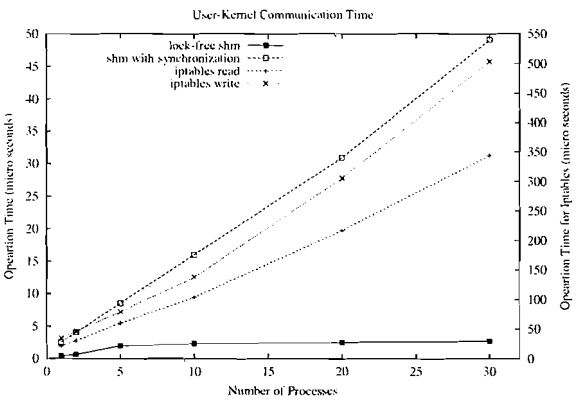

Figure 9: Effect of increasing the number of concurrent processes on user-kernel communication time.

lock-free shared memory increases only marginally when the number of concurrent processes are five and then remains approximately the same. This implies that number of context switches while the operation is executing increase negligibly with increasing number of concurrent process. On the other hand the completion time for user-kernel communication using shared memory with synchronization increases rapidly. The measurements for Iptables (plotted on the right hand y-axis) shows a similar trend. The plot shows that the time for completion of communication operations based on system call increases as the number of concurrent processes increases. (Note that these values do not represent the CPU time taken by the operation).

Next we repeated the same experiment using a different traffic generator, which produces 64 byte packets at 5000 pps. This traffic generator uses `usleep()` because the inter-packet generation time is large and does not require much accuracy. The results for this experiment are provided in Figure 10. The results show that for up to 10 concurrent traffic generators the completion time for lock-free algorithm increases negligibly, while the increase for shared memory with system call for synchronization increases slightly. Nevertheless, its completion time magnitude is 10 times higher as compared to the lock-free algorithm. One should note that, for up to

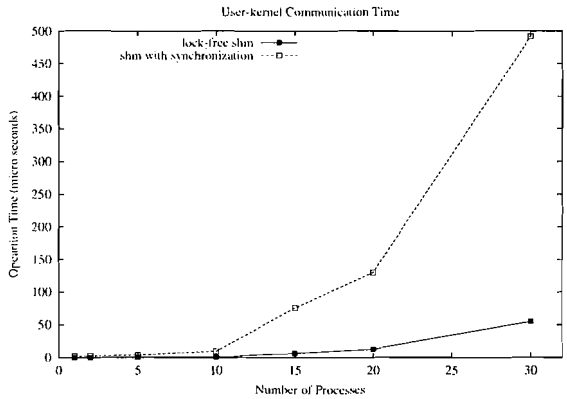

Figure 10: Effect of increasing the number of concurrent processes, which use sleep, on user-kernel communication time.

10 concurrent processes, both algorithms performed better than in the previous experiment, which is a result of lesser number of ready processes and context switches, because the traffic generators are mostly sleeping. However, when the number of concurrent traffic generators increase to 15 the cost jumps. This is because with 15 traffic generators there are always processes ready to run and more sleep timers go off as the ready queue builds up (we call it scheduling collapse). The the overhead of sleep timers going off and their scheduling becomes very high. The number 15 is a function of the rate at which the traffic generator generates packets, if the packet generation rate is higher, i.e., inter-packet generation time is smaller, similar results would be seen with lesser concurrent processes. The result of this scheduling collapse is reflected in the jump of completion time for shared memory with system call for synchronization to around 75 μs which is higher than the case with 30 concurrent sessions in the previous experiments. The lock-free shared memory Q-Pod is also affected by this scheduling collapse and performs worse than for the previous experiment. However, the key observation is that lock-free shared memory algorithm shows a cost almost 10 times smaller than the case where system call is used, irrespective of the number of concurrent processes. This is consistent with results from the previous experiment where the cost for lock-free shared memory Q-Pod was around 5 to 10 times smaller.

Given this low level evaluation of the effect of using system calls—(1) increased CPU time usage and (2) increased possibility of context switch during the operation, we next evaluate how this adversely effects the performance and functionality of Q-Pod mechanisms.

4.2.1 User-Kernel Communication Rate

The only difference between policy updation and reading measurements is that one involves a user-space thread writing to the shared memory and the other involves reading from it. As remarked earlier, these two measurements do not differ much. Hence, here we present only our results for reading from the shared memory. We also omit the results for Q-Pod version using read and write system calls for user-kernel communication.

To evaluate the effect of using lock-free shared memory and system call based synchronized shared memory for user-kernel communication, on the performance of Q-Pod, we performed an experiment where a Q-Pod's thread reads from the shared memory in an infinite loop. We vary the load—number of concurrently running UDP CBR traffic generators (generating 64 byte packets at an aggregate rate of 50 kpps). The user-kernel communication rate of the two versions of Q-Pod, in terms of the number of read operations, performed by the Q-Manager thread, per second is studied. The results are meaningful in contrasting how the use of a system call effects the rate at which the Q-Manager can access the data logged by the Q-Driver, and the effect of concurrent

load on its ability to do so. However, the traffic data collected by the Q-Manager is not meaningful in this experiment (and we will investigate the effect on traffic measurements latter). The plot in Figure 11 shows our results. For a single process load Q-Pod using lock-free shared memory is able to perform almost double the

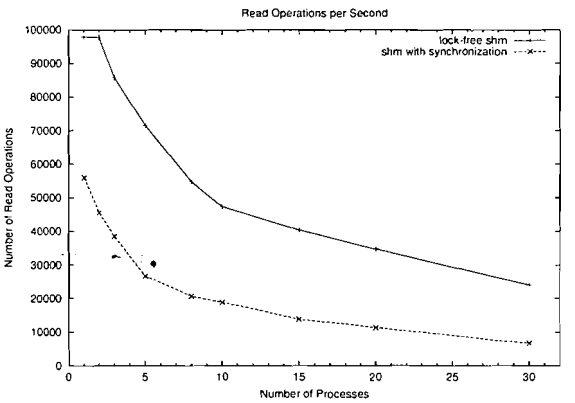

Figure 11: Comparison of user-kernel communication rate.

number of operations as compared to its counterpart using synchronization. When the load increases to two process, Q-Pod with lock-free shared memory is able to perform almost the same number (slight degradation) of operations because of the construction of the Q-Manager thread. It contains an infinite loop, within this loop is another loop which traverses a list (maintained by the Q-Manager) of currently active managed sessions, and reads information about each session. With a single session the inner loop runs once returning to the outer loop, and then again starting the inner loop afresh. With two sessions the inner loop runs twice, for each run of the outer loop. This makes use of instruction and data cache resulting in two shared memory access being performed in a smaller time—enough benefit to result in the same number of read operations even with lesser CPU time (roughly speaking $1/3^{rd}$ with two traffic generators vs. $1/2$ with one traffic generator) allotted to this Q-Manager thread by the scheduler. However, this benefit does not manifest it self as the load keeps increasing the CPU share of the Q-Manager keeps decreasing. Certainly, the version of Q-Pod using shared memory with system call for synchronization enjoys the same design. However, the decrease in the number of read operations due to its slow nature overshadows this benefit. The performance of Q-Pod using shared memory with synchronization rapidly deteriorates. This trend follows as load increases. On the other hand the performance of Q-Pod using lock-free shared memory degrades slowly. A comparison of the values for the two versions of Q-Pod show that Q-Pod with user-kernel communication based on lock-free shared memory outperforms its counter part by at least a factor of two—when the load is 1—and factor of about four—when the load is 30.

We repeated the above experiment using the second type of traffic generator which uses `usleep()` and sends packets at only 5000 pps. Our results for this case are shown in Figure 12. Once again up to 10 concurrent session both versions of Q-Pod performed better than in the previous case, owing to the fact that Q-Manager thread gets more CPU allocation while the traffic generators sleep. The differences in the magnitude of the rates for the two versions is consistent with our previous experiment—Q-Pod with lock-free shared memory performs at least twice and up to five times better than its counter part. One interesting observation from this plot is the initial increase in rate of read operations as the number of sessions increase for both versions of Q-Pod, which is consistent with the design description of Q-Manager thread that we gave earlier. However, once again we note that the slower version using system calls is not able to take much benefit from this design.

Consider writing of a measurement by the Q-Driver as an *event*. One of the key disadvantages of slower user-kernel communication rate, in a *polling* scenario as described above, is that each event may not be noted individually by the Q-Manager. Hence, the Q-Manager will notice that multiple events have occurred between

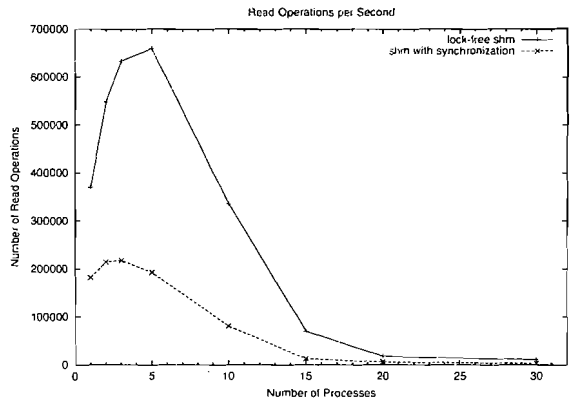

Figure 12: Comparison of user-kernel communication rate, using traffic generators which employ sleep.

two consecutive measurements (read from shared memory) for a given session. We call this *accumulation of events*. The higher this number the less responsive the Q-Manager will be. For example, it may effect the ability of Q-Pod to keep accurate track of a moving average of a network session's throughput. In short in certain scenarios we may want the number of accumulated events to be small. To study the performance of the two versions of Q-Pod using this metric, we setup an experiment such that there is one UDP receiver to which 64 byte packets are sent at 50 kpps from an external host. Other than this receiver traffic generators sending 64 byte packets at an aggregate rate of 50 kpps serve as "load" on the end system. This was done so that the rate of generation of event—message written by the Q-Driver indicating a packet reception—is independent of the CPU scheduling occurring on our test end system. The plot in Figure 13 shows the distribution of the number of accumulated events observed for both versions of Q-Pod when the load (including the receiving process) was 10. (The few, $< 0.5\%$, cases where Q-Pod was context switched out for an extended period, leading to greater than 100 accumulated events are not shown). The plot shows that Q-Pod using lock-free shared memory based user-kernel communication observed a queue length of ≤ 2 in 90% of the cases. While Q-Pod using synchronization based shared memory observed queue lengths of ≥ 4 in 80% of the cases and ≥ 10 in 10% of the cases—implying both higher magnitudes and variations.

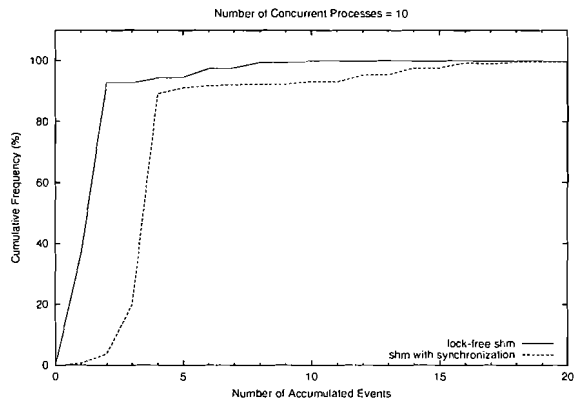

Figure 13: Cumulative frequency distribution of queue lengths.

4.2.2 Evaluation of Footprint

The user-kernel communication required by Q-Pod exerts an overhead on the CPU of end system, in terms of consuming CPU time. Using our base results from Table 5 it is easy to see that using lock-free shared memory exerts a lower footprint as compared to using synchronization based shared memory, if the number of communication operations are the same. The results in Section 4.2.1 indirectly confirm this claim—given the same CPU time proportion, Q-Pod using lock-free shared memory for user-kernel communication, is able to perform higher number of communication operations. Following this argument, if the the communication rate, of the two Q-Pod versions (lock-free and synchronization based) using shared memory, is the same, Q-Pod using lock-free shared memory should exert a lower footprint resulting in better performance of the other processes running on the end system. For example, theoretically, if the lock-free shared memory operation takes 0.5 μs and synchronization based shared memory consumes 2.5 μs and both perform communication at the rate of 50,000 operations per second, the per second CPU overhead of the former is $50000 * (2.5 * 10^{-6} - 0.5 * 10^{-6}) =$ 100 ms less than the latter. However, practically this can not be achieved because controlling the rate implies introducing overheads due to calculations and more importantly due to increased context switching—to allow other applications to use the saved time the communication thread must yield the CPU. The following results reflect the magnitude of gain achieved when the communication rate is the same for both Q-Pod versions, and qualify the effect of this gain in terms of end system performance.

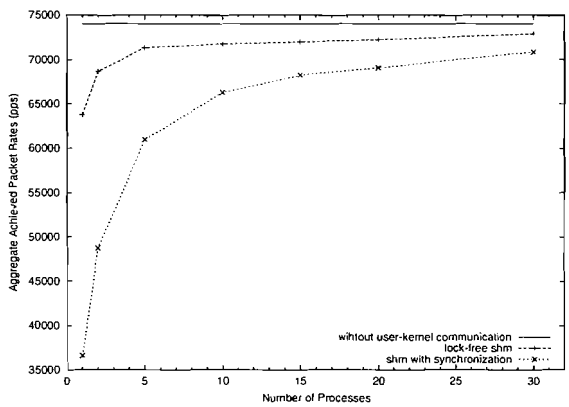

Figure 14: Gain due to lower footprint of lock-free shared memory based user-kernel communication, in terms of achieved aggregate packet rates.

We modified lock-free shared memory based version of Q-Pod such that it achieves the same rate as reachable by shared memory with synchronization based Q-Pod running at full throttle. This was achieved by making Q-Pod sleep, but at the same time maintaining the correct rate uniformly. The experiment was performed with varying load (concurrent UDP CBR processes generating 64 byte packets at an aggregate rate of 75 kpps). A receiver on an other host receives these packets and shows the achieved packet rates. If the processes are run without Q-Pod they exactly achieved the configured rate. Figure 14 shows our results. Using aggregate achieved packet rates as the performance metric, and given a user-kernel communication rate, lock-free shared memory access has a lower overhead, allowing more processing time for the load processes. Our results show a gain of almost 75% for the single process case. Even for the case with 30 load processes the achieved packet rate is almost 3000 pps higher than its counterpart, which is a considerable amount.

Our results demonstrate the high performance of user-kernel communication using our lock-free shared memory scheme, in terms of scalably affording higher rate of communication, timely event processing and lower footprint.

4.3 Packet Interception and Processing Footprint

Q-Driver introduces an overhead on every outgoing and received packet, because of interception and processing. We performed low level measurements of the processing that occurs with in the interception functions to evaluate this. This is measured using the `rdtsc` command, giving nanosecond granularity and average values using 1000 measurements are used. In Table 6 we show the measurements for IOP (outgoing packets). The processing cost involved are (1) extracting header fields, (2) classification as filtered managed or unmanaged session, and (3) in the case of managed sessions updating TOS fields, updating IP checksum and recording traffic measurements. Our measurements at the IRP had similar results, except the total cost of processing managed packets was 495 ns, which is lower than that in IOP—in IRP the IP TOS field is not marked and checksum need not change.

Table 6: Packet processing cost (in nanosecond) for the filtered, managed and unmanaged packets, in IOP.

Overhead	filtered (ns)	managed (ns)	unmanaged (ns)
Pkt. parsing	65	65	65
Classification	40	350	670
Updations	-	210	-
Total	105	625	735

Table 7: The number of sessions, out of a total of 85, which required the given number of retries to create an entry and the time measured in IOP, required to search that entry.

Retries	Ses.	Time (ns)
Zero	80	350
One	3	380
Two	1	445
Three	1	490

Recall, that our classification scheme for managed sessions involve using multiple hash functions to search an entry in the Session Table. In our implementation we have used a Session Table of size 97 and 4 hash functions. Given 85 concurrent clients (which get local port number assigned by the OS assigned), the distribution of retries required to successfully create an entry is shown in Table 7. It takes greater time to classify a packet who session entry was created after multiple probing. The different classification times (measured in IOP) are also show in the table. Thus, the calculation provided in Table 6 hold for about 94% of the sessions under high concurrency.

The results presented show the overhead of processing within Q-Pod's packet interception functions. However, it does not show the overhead of inserting a module in the packet flow path, i.e., the cost of handover of packet to the Q-Driver's interception functions. To calculate this overhead without changing legacy kernel code we setup the experiment as show in Figure 15. We measure the round trip time (RTT) of a UDP packet, which is received and echoed back to the sender, using a Linux host (B) with and without Q-Pod. UDP packet of size 64 bytes was used, to minimize the contribution of cost of transmission on Ethernet.

All else being equal the difference in the RTT for results with and without Q-Pod is attributed to Q-Driver per-packet overhead. The packet RTT is measured at Host A using `tcpdump` [6] which gives microsecond granularity. The time per-packet shown is averaged over 10,000 packets. Table 8 summarizes our results. Using our results for processing inside IRP and IOP we calculate the total handover overhead—sum of handover for received and outgoing packets.

The results show that the packet interception and processing cost for Linux is 1.23 μs, which implies 1.73%

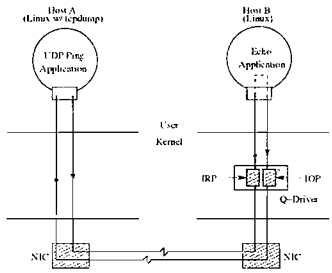

Figure 15: Experimental setup for packet interception and processing cost evaluation using RTT. Host B shows the Q-Driver installed in the kernel.

Table 8: Round-trip times (in microseconds) for Linux, with and without Q-Pod and the breakdown of this overhead.

Item	Time μ
RTT with Q-Pod	72.12
RTT without Q-Pod	70.89
Difference	1.23
IRP cost	0.495
IOP cost	0.625
Handover cost	0.11

overhead. This cost is negligible given that transmission delays are usually much higher (e.g., due to distance, queues at routers, and larger packet sizes) than in our experimental conditions and fluctuate over a range which is at least an order of magnitude higher than this cost.

4.4 Performance Evaluation of Dynamic Session Discovery

DSD cost depends on the event notification from the kernel to user space, mapping sockets to applications, admission control, creating entry in the shared tables and deleting the entry from the EDQ. This is a one time cost per each new session. Table 9 summarizes our measurements for DSD taken using `rdtsc`, where one timestamp is taken before new session notification and EDQ insertion takes place in the Q-Driver and the other timestamp is taken after deleting the entry from the EDQ in the SDM thread of the Q-Manager. DSD cost depends on the number of sessions already running on the end system—lookup of running processes and open sockets becomes

Table 9: Average time (in milliseconds) for dynamic session discovery of UDP and TCP sessions on Linux.

Load	Linux Q-Pod	
(Sessions)	UDP (ms)	TCP (ms)
1	1.0	3.2
10	1.1	4.4
20	1.4	4.8
30	1.6	5.5

slower. In the table we show how DSD scales with load. A further dissection of the cost showed that event notification, on the average over 100 observations, takes 312 μs, and this average increases marginally as load increases. Static admission control scheme, creation of entries and EDQ insertion and deletion are all memory access and their contribution to this cost is low. The major bulk of DSD cost arises from socket-to-process id mapping. We perform the mapping in user space. We opted not to use Linux kernel space implementation, which is usually faster due to direct access of process and connection tables, to maintain uniformity and platform independence in Q-Pod's design—operating systems with layered architecture (e.g., Windows) do not allow process table look up in low level network drivers i.e., Q-Driver. The mapping involves a scan of the process tables and open connections using `/proc` file system. Increasing the number of processes and sessions increase this cost because there are more entries which need to be scanned. The interface provided by `/proc/net` scans the UDP and TCP hash tables maintained by the Linux kernel for open connections. The hash table for TCP connections is larger, on Linux 2.4.21, than the UDP table. This results in a higher cost for mapping TCP sessions.

Due to our semantics to strictly adhere to QoS policies which should be enforced on every packet for a given session, the first packet of the session is queued and is released when DSD completes. The effect of this cost on performance can be analyzed in the light of different workloads. For UDP workloads, the first packet will be delayed by this time and any packets received during this interval will be dropped—which depends on the packet rate. Multimedia is an important example for use of UDP. For example, VoIP with 10 ms of audio sample per-packet, or 100 packets per second (pps), will have no packet drops. As the DSD delay is only at session startup, it will have un-perceivable impact on multimedia sessions which are usually long lived. For local area UDP traffic e.g., Domain Name System (DNS) traffic this overhead can be avoided by categorizing the DNS traffic as filtered and hence avoiding DSD. For TCP sessions, the session connection setup will be delayed by this time. This is a one time overhead and thus needs to be analyzed relative to the session life time. This is evaluated in the context of Internet workloads in Section 4.6.

4.5 Q-Pod Evaluation For Multimedia Workload

Multimedia applications are characterized by high packet rates and need for low latency, jitter and packet loss rate. We have already shown that the effect of Q-Pod on latency is small, Q-Pod does not introduce jitter, and that packet losses that can occur once at the start of a new session due to DSD are minimal. However, there is a need to qualify the effect of packet processing cost—however small—when the packet rates are very high, e.g., in the case of multimedia content distribution servers with several multimedia streams.

The performance of Q-Pod under high packet rates can be used to extrapolate the overhead of Q-Pod for very high load multimedia applications. We characterize this performance by the maximum loss free packet rate (MLFPR) achievable by an application running on the end system (via an idle 100 Mbps switch). MLFPR is defined as the UDP packet rates observed at the receiver when a UDP packet generator sends packets at the fastest possible rate without incurring packet losses (these may occur due to packet queues, in the kernel, being filled because of the application generating packets at a higher rate than the rate at which they can be transmitted). MLFPR depends on the size of the packet. For larger packets the 100 Mbps Ethernet bandwidth ceiling limits the packet rate, on the other hand, for smaller packets the CPU of the sender's end system limits the packet rate (given that packet handling at the receiver is triggered by interrupts).

We set up an experiment to evaluate MLFPR for different packet sizes for Linux using a UDP traffic generator which sends packets of a given size as fast as possible without incurring packet loss. Effects due to fragmentation are avoided by limiting the packet size to the MTU size (1500 bytes).

The plot in Figure 16 shows the MLFPR (y-axis) vs packet size (y-axis), with and without Q-Pod. The plot clearly shows that the footprint of Q-Pod's packet processing is not significant for packet rates up to 80 kpps

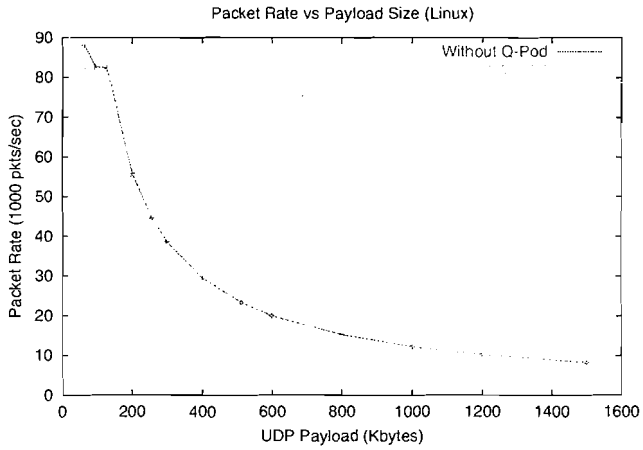

Figure 16: Maximum loss free packet rate measured at the receiver vs packet size for Linux, with and without Q-Pod.

(1000 packets per second). We also notice that Linux without Q-Pod reaches its limit around 80 kpps as well—the packet rates for a packet of size 128 and 96 is almost the same. However, with a packet of size 64 bytes the packet rates jumps to 88 kpps—this is because IP stacks are optimized to handle 64 bytes packets (the minimum allowable under Ethernet, excluding FCS). On the other hand with Q-Pod the rate remains saturated at around 82 kpps, because of the processing overheads depend on packet rates and not sizes, giving a drop of 6.5% under this worst case.

Realistically, 64 byte packets (i.e., a maximum of 22 bytes for payload and application level headers, e.g., RTP) are rarely used for multimedia because of high network overhead per packet. For higher size packets, Q-Pod's low footprint results in an insignificant performance-overhead even for multimedia servers with several streams which aggregate to packet rates of up to 80 kpps.

4.6 Q-Pod Evaluation For Internet Workload

Internet workload is characterized by the bulk of TCP sessions being short and a few sessions being very long [31]. The few long TCP sessions, however, take up most of the bandwidth. To evaluate the overhead of Q-Pod under Internet workload conditions we study the TCP session completion time for varying TCP payloads.

For this experiment we use a simple TCP file transfer client/server application. The completion time is measured as the time required to connect to the server, completely send a payload to the server, receive a message from the server to mark completion, and closing of the TCP session. The TCP server is run on a Linux machine without Q-Pod. The TCP client is executed on an end system running Linux with and without Q-Pod. Figure 17 shows our measurements. The payloads are varied from 50 kbytes to 5 mbytes. The x-axis is the payload size. The left y-axis is the completion time in milliseconds. From the plot we observe that Q-Pod results in an almost constant overhead of about 7 milliseconds with respect to the completion time. This overhead is due to the cost of DSD and its impact on packet scheduling. The overhead is represented as a percentage of the completion time without Q-Pod in Figure 17 using the right y-axis. For short-lived sessions the overhead is high—around 150%, but for long-lived sessions the overhead becomes increasingly insignificant. However, it should be noted that this experiment was performed on machines directly connected via a switch, and hence session life time do not take conditions of long haul WAN environments. As the overhead is in the order of few milliseconds it may not be perceivable by the user and blend with other "noise factors" such as network latency

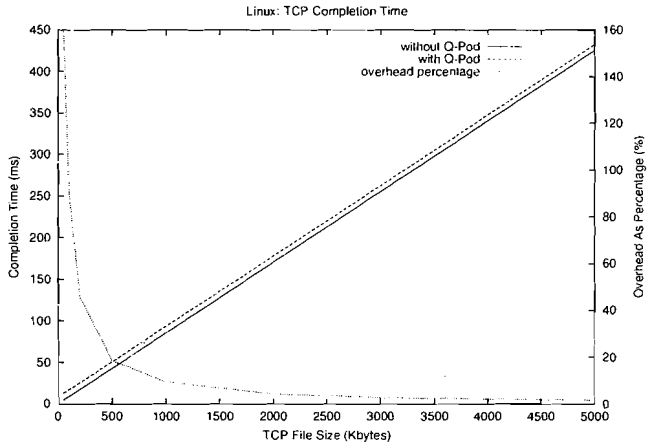

Figure 17: TCP session completion times (in milliseconds) for varying payloads, and the overhead of Q-Pod as a percentage.

and packet losses (and hence, delays due to retransmissions). In fact even on machines connected via switch, a variability, in session lifetimes, of around 2 ms is observed.

In the real world TCP sessions may face congestion, which elongates the session lifetime. In such cases, the lifetime of even small sessions will be higher reducing the relative overhead of Q-Pod. Furthermore, the purpose of Q-Pod is to provide QoS support for legacy applications. QoS support, presumably, implies networks where there exists contention for network resources and hence congestion. In such cases the benefit of Q-Pod in achieving QoS for legacy applications outweighs the overhead it incurs due to DSD. Following this argument a TCP session may experience a shorter lifetime, even after incurring DSD overhead, due to the QoS it receives. Nevertheless, it must be stated that such an advantage is a function of the QoS infrastructure (router QoS mechanisms), and the role Q-Pod plays is to enable legacy applications to utilize the infrastructure. The experimental setup and the results to illustrate this is shown in Figure 18. Q-Pod results in significantly lower completion times by benefiting from the traffic prioritization, based on TOS labels, at the routers.

In summary, Q-Pod introduces a fixed overhead on TCP session completion time. However, the small absolute magnitude of this overhead suggests that in WAN environments this overhead is insignificant. Provided that Q-Pod enables QoS for the traffic, the QoS advantage more than compensates for this overhead.

4.7 Q-Pod Scalability Under Heavy Workloads

To demonstrate the scalability of Q-Pod we provide a performance evaluation of Q-Pod under heavy workload. In this experiment we use UDP CBR traffic generator-receiver pairs running as Q-Pod managed applications on two end systems. Our objective is to evaluate if the traffic generators are able to perform seamlessly—achieve throughput equal to the configured data rate. The load on the end system is defined by the number of such applications that are run concurrently, each sending data at a high rate. Recall that dynamic session discovery, per-packet interception and processing, and logging traffic measurements to disk, make up the overhead introduced by Q-Pod.

The parameters for this experiment are as shown in Figure 19. The plots in Figure 20 show a trace of the lifetime of the sessions for this configuration and the throughput-to-offered load ratio for each application.

All 75 application sessions were able to achieve the target throughput resulting in a ratio of 1. In our experiment the max load was sustained for around five minutes. Increasing this does not effect our results, however, increasing the number of concurrent sessions results in a few sessions not achieving a ratio of 1. Similar

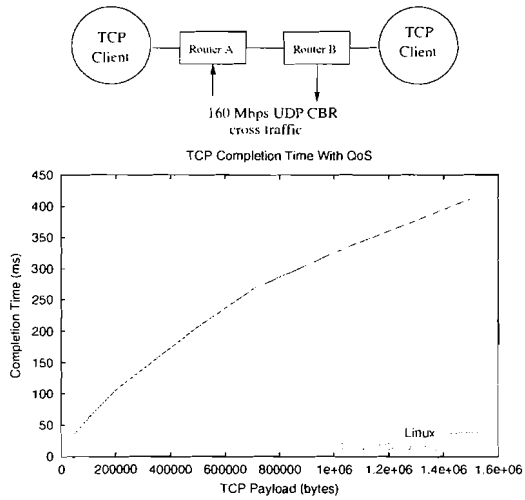

Figure 18: (Top) The TCP client and server are connected via two routers. The 155 Mbps OC3 link between the routers is congested using 160 Mbps UDP CBR cross-traffic. (Bottom) TCP session completion time under congestion. Q-Pod endows QoS to the session and hence achieves lower session completion times.

effect is seen when Q-Pod is not installed on the end systems, implying that the reason for this is because of CPU scheduling and packet transmission limitations and not Q-Pod. Realistically, 75 concurrent sessions on a single machine, launched with a 2 second inter-arrival time, each sending 1 Mbps is a very high load and not representative of the typical workload experienced on an end system. The stress test demonstrates that Q-Pod has a wide operating range and is scalable.

4.8 Analyzing Windows XP vs Linux Q-Pod Performance

Q-Pod's performance is intricately related to the platform on which it is implemented. Using a comparison of measurements on Windows XP and Linux Q-Pod we highlight the dependence of Q-Pod's performance on the underlying platform. We present only the results which are different for the two platforms, e.g, the performance of lock-free shared memory based user-kernel communication on both the platforms is similar and thus is omitted.

Differences in driver models, network subsystem and packet structure: Windows XP has a layered kernel architecture as compared to the monolithic kernel of Linux. In general the layered architecture exerts higher processing overhead in the packet forwarding path. Thus, repeating our UDP ping experiment showed a higher RTT for Windows XP. The overhead of Q-Driver, in the packet path was also higher. This is because of two reasons: (1) the packet parsing cost to get header fields is higher because of NDIS packet structure (Figure 7 shows the packet) and packet handling rules, and (2) because the Q-Driver is wrapped by NDIS, the overhead of inserting the driver in the packet forwarding path is also higher as compared with Linux where it is a simple function call from within the IP stack. Table 10 summarizes Windows XP measurements and comparison with Linux. As expected, we also observe a difference in the ability of Windows XP to handle packets at a high rate. The plot in Figure 21 shows the impact of higher processing cost in Windows XP. Nevertheless, the overhead of Windows XP Q-Pod is also small, e.g., the overhead on MLFPR for a packet of size 96 bytes on Windows XP is 8%.

Difference in available APIs/system calls: These differences are highlighted by the DSD cost analysis. Windows XP allows us to use an API for kernel-user events which is more costly (589 μs vs 312 μs for Linux), because of the layers of the I/O subsystem that such events have to traverse and differences in process manage-

UDP CBR CONFIGURATION:
Lifetime = 470 s
Data rate = 1 Mbps
WORKLOAD CONFIGURATION:
Inter-arrival time = 2 s
Number of end systems = 2
Max load = 75 (concurrent sessions)
Max load reached at time = 150 s
Max load sustained for time = 320 s
Total data rate at max load = 75 Mbps
METRIC:
Throughput-to-offered load ratio
Offered load: CBR UDP configuration
Throughput: measured at run-time by Q-Driver

Figure 19: Configuration for the scalability test.

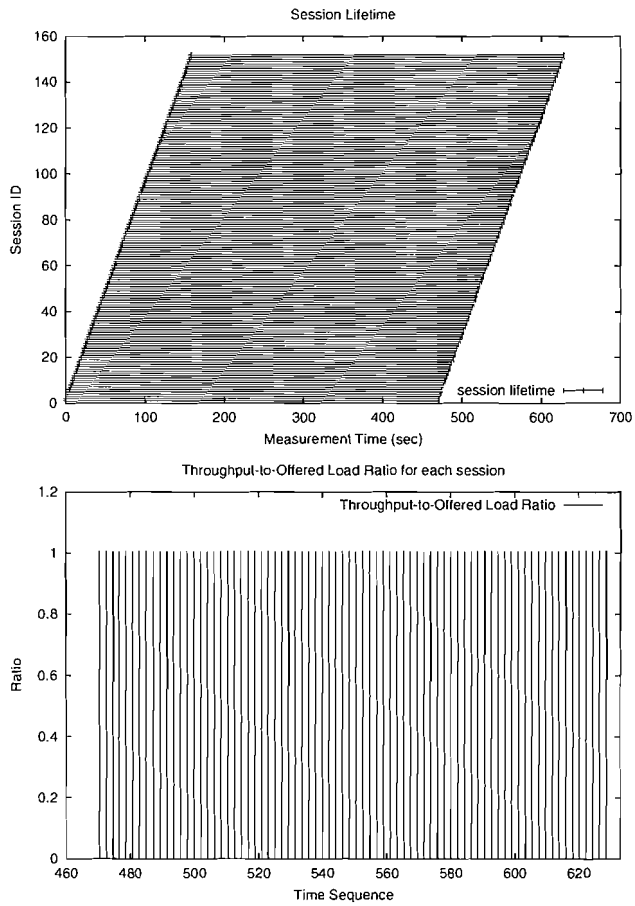

Figure 20: The trace of the session lifetimes and the ratio of throughput-to-offered load. The plot shows that throughput equals offered load for all 75 sessions.

ment. In Linux we implemented the events by affecting the scheduling of SDM directly. Similarly, in Windows XP we use APIs to perform session to process mapping, as compared to the lookup of `/proc` file system in

Table 10: Comparison of packet processing overhead

Item	Windows XP (μs)	Linux (μs)
RTT w/ Q-Pod	91.09	72.12
RTT w/o Q-Pod	87.30	70.89
Difference	3.79	1.23
IOP	1.871	0.625
IRP	1.241	0.495
Handover	0.68	0.11
Pkt. parsing	0.755	0.065

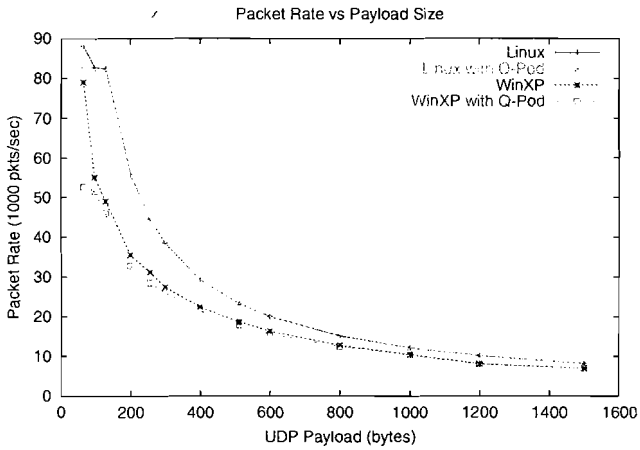

Figure 21: Maximum loss free packet rate for both Windows XP and Linux with and without Q-Pod.

Linux. Both these methods have their intrinsic costs which are similar. However, Windows XP API results in a slightly higher cost when load increases. As is the case with Linux the cost of mapping a TCP session is higher than that for UDP, on Windows XP.

4.9 Summary

In this section we have demonstrated the high performance and lock-free shared memory base user-kernel communication in contrast with other user-kernel communication modes. Low-level and traffic level measurements show that the overhead of Q-Pod on the network traffic is small. Q-Pod has a low footprint under varying and stress workload which highlight its wide operating range and scalability. A comparison of results between Linux and Windows XP show the dependence of Q-Pod performance on the underlying operating system. The low footprint for both the platforms demonstrate the efficacy of Q-Pod design.

5 Network QoS Benchmark

Q-Pod integrates with QoS mechanisms exported by legacy routers to facilitate scalable end-to-end QoS. In this section we demonstrate that Q-Pod enables "turn-key" QoS support to legacy applications running on legacy operating systems. Q-Pod enforces the QoS policy in the form of transparent IP TOS field marking on the network sessions of legacy applications, achieving the user-specified QoS requirement capitalizing on QoS

mechanisms, in our case studies *Assured Forwarding* (AF) per-hop behavior (PHB), enabled on legacy routers. Most routers also allow providing PHB for default (e.g., un-marked) traffic relative to marked traffic, allowing incremental deployability of Q-Pod—all end systems need not have Q-Pod, only those where users require QoS.

In this section we show end-to-end network QoS performance for voice-over-IP (VoIP), message passing interface (MPI) based grid computing and H.323 compliant real-time streaming multimedia content distribution network (CDN). These are examples of already existing, and popularly in demand, QoS sensitive enterprise level applications Each of these experiments are performed using legacy applications on Linux and Windows XP platforms.

5.1 Setup

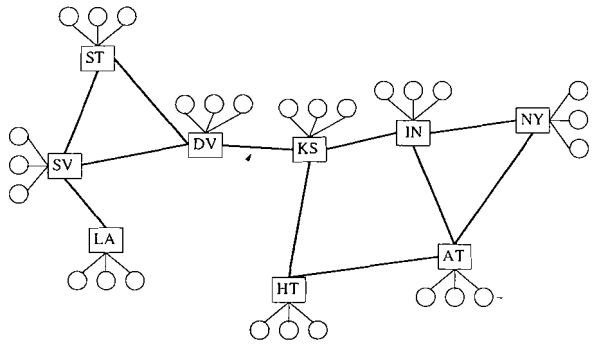

Figure 22: Network testbed consisting of 9 backbone Cisco routers connecting 30+ end systems.

For our network experiments we use the testbed depicted in Figure 22. This testbed consists several end systems connected via 100 Mbps Fast Ethernet to the routers. The 9 routers in our testbed are CISCO 7206VXR with NPE400 and NPE300 processing units, running CISCO IOS 12.2. Each router has 2-3 OC-3 (155 Mbps) Packet-over-SONET (POS) links for connectivity with other routers. The routers' physical connectivity follows that of Abilene/Internet2. The cross-traffic is sent over the link from Kansas (KS) to Denver (DV) router.

The self-similar bursty nature of network traffic, resulting from heavy tailed TCP file transfers and bursty session inter-arrival times, is an ubiquitous phenomenon observed on real-world networks. It is this bursty nature which makes it difficult to provide QoS to quality sensitive applications even on networks which are amply over-provisioned with respect to average offered load. Thus, we use cross-traffic exhibiting bursty nature as our cross traffic. If routers are indifferent to the QoS requirements of network sessions across them, this bursty traffic results in unacceptable quality for QoS sensitive applications such as the ones we consider in this section. Q-Pod transparently enables QoS support for such applications integrating differentiated service already supported by commodity routers to achieve scalable end-to-end QoS. We generated the cross-traffic using a UDP traffic generator giving the trace with 2 second aggregation as shown in Figure 23. The 2 second aggregation implies that each impulse is maintained for 2 seconds. This trace was generated using *Pareto* distribution, $F(x) = P[X \leq x] = 1 - (k/x)^{\alpha}$, with $k = 40$ Mbps and $\alpha = 1.05$. We capped the bursts at 155 Mbps. The result is bursty traffic as the trace illustrates.

The routers were configured with AF PHB using weighted random early detection (WRED) as supported on CISCO IOS 12.2. This uses differentiated services code point (DSCP) in the first six bits of the TOS field. We used a single AF class, AF1 with three drop precedences (DP). The parameters used are {min threshold, max threshold, drop probability}: AF11 = {50, 70, 1/40}, AF12 = {5, 40, 1/10} and AF13 = {3, 4, 1}, in order of increasing DP.

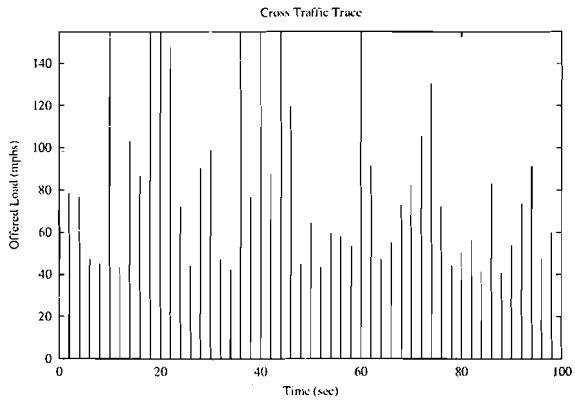

Figure 23: "Bursty" cross traffic, generated using Pareto distribution with $\alpha = 1.05$ and $k = 40$ Mbps, capped at 155 Mbps.

The end systems, running Linux 2.4.2x and Windows XP, have Q-Pod installed and running on them. The legacy applications are simply started using a command-line version of Q-Interface e.g.,

```
qInterface -app ''name <parameters>'' -qos HIGH
```

Next, the Q-Manager and the Q-Driver dynamically discover network sessions of the application at run-time and enforce the TOS marking, transparent from the user and the application.

5.2 VoIP

With the ever increasing popularity of VoIP, there is also a growing concern for providing QoS for VoIP to match toll quality voice transmission. In an enterprise environment with traditional phones replaced by VoIP one could imagine several hundred voice calls streaming across routers. For this case study we simulate 640 G.729 VoIP calls using a traffic generator, with the configuration show in Figure 24. We chose G.729 as it is

Protocol	G.729 over RTP
Voice payload	20 Bytes (2 samples)
Packet size	ETH+IP+UDP+RTP+PAYLOAD+FCS $= 14+20+8+12+20+4 = 78$
Required packet rate	50 pps (or 31.2 Kbps)
Number of flows	640
Total packet rate	32 kpps (or 19.97 Mbps)
Traffic generator	UDP CBR
Packet size	78 Bytes
Offered packet rate	31.9 to 32.5 kpps
Offered data rate	19.90 to 20.28 Mbps

Figure 24: VoIP traffic setup: UDP CBR traffic generator simulating 640 G.729 flows.

the ITU [3] recommendation for VoIP. The traffic generator is treated as a legacy application by the Q-Pod and is executed using the Q-Interface. The traffic generator is executed at a host connected to the IN (Indianapolis) router, while the receiver is connected to the DV router. The measurements of the offered and achieved packet rates are collected by Q-Pod. To switch on QoS, AF is enabled at KS and DV routers, and Q-Pod on the end

system performs transparent TOS marking endowing QoS to the voice traffic. Q-Pod sends voice traffic to AF11 and the cross-traffic to AF13. To disable QoS, AF is disabled.

Our measurements for the offered and achieved packet rates for the two scenarios (with and without QoS) are shown in Figure 25. The packet rate trace for the sender and the receiver, for the case with QoS enabled, reflect end-to-end QoS with immunity from the cross-traffic. Also note that the queuing delay at the routers in bursty periods is not large enough to be noticeable as a drop in packet rates. While in the case of best-effort service the receiver observes sharp drop in packet-rates, corresponding to the packet drops due to contention with peaks in the cross-traffic.

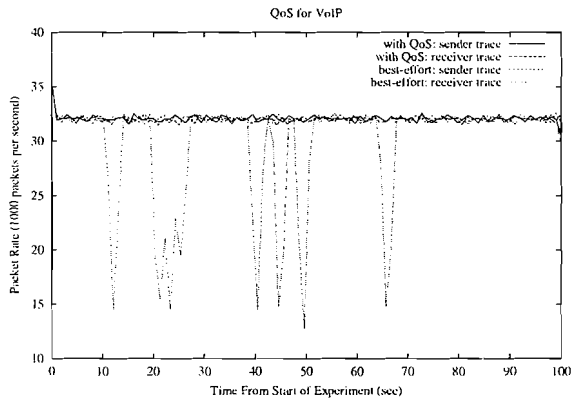

Figure 25: Traces for VoIP traffic. When VoIP traffic is sent using best-effort service, bursty cross traffic results in packet rates reduced by almost 50% (blind fairness!) as can be seen.

5.3 MPI Based Grid Computation

MPI is a popular interface for executing heavy computations in a distributed environment, e.g., large scale simulations. However, the communication overhead between participating end systems is one of the bottlenecks in this distributed environment. In some cases a dedicated network may be used to connect end systems. However, this leads to increased costs, which make using already existing network infrastructure attractive for organizations and educational institutions. On a best-effort network existing traffic may slow down distributed computation due to increased communication overhead. Q-Pod can endow QoS to such critical legacy applications resulting in reduced communication overhead, and thus smaller completion times. To illustrate this we use a Dassfnet [1] based simulation which uses MPI. We configured two end systems (one connected to IN and the other to DV) to participate in a short simulation of "worm propagation on a 300 node network"[9]. Q-Pod's dynamic sessions discovery is able to detect the run-time negotiated sessions between participating hosts, and the transparent policy enforcement on the network sessions leads to end-to-end QoS for the TCP sessions in which the distributed hosts engage. The effect of this manifests itself in a smaller simulation completion time. Given cross traffic between the routers we ran the MPI application with and without Q-Pod. In the case with Q-Pod, AF is enabled on the routers and MPI traffic is sent to AF11, while the cross traffic goes to AF13. Our results, shown in Figure 26, reflect the efficacy of Q-Pod. Q-Pod endows QoS resulting in saving of up to 18%, on the completion time.

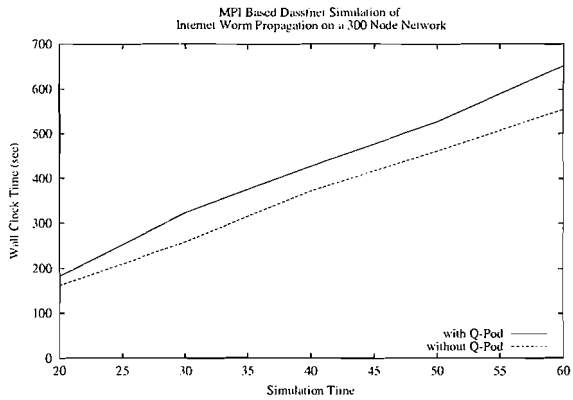

Figure 26: Comparison of simulation completion times under cross traffic, with and without Q-Pod support.

5.4 Real-Time Multimedia CDN

Real-time multimedia CDN is popularly in use by educational institutions for out-of-classroom live lectures. It is also gaining popularity with service providers, who intend to provide live media (e.g., cable TV) as a value-added service to their customers. H.323 is popular standard for managing and transmitting real-time multimedia. In this case study we use legacy H.323 compliant applications. We use *NetMeeting* which is provided with Windows XP, and *OpenPhone* [5] also installed on Windows XP as the clients of real-time multimedia. We use *OpenMCU* [5] on Linux, as a Multi-point Connection Unit (MCU) which receives real-time cable TV transmission from a dedicated encoder and transmits it to subscribing clients. The MCU is connected to the Atlanta (AT) router. The NetMeeting client is connected to DV and the OpenPhone client is connected to the Los Angeles (LA) router. Q-Pod is installed on these three machines. We implement a simple destination address based admission control on the Q-Pod running on the MCU end system, such that it sends traffic destined to NetMeeting end system using AF11 to provide gold quality, while the traffic destined for the OpenPhone end system is transparently sent to AF12, to provide silver quality. The cross traffic on the KS to DV link is sent to AF13. As the traffic stream in this experiment peaks only up to a few Mbps, we added additional 30 Mbps UDP CBR traffic sent to each of AF11 and AF12. This is a practical example where Q-Pod is used to provide differentiated quality for clients, by endowing QoS capability and inserting admission control intelligence transparently to the legacy H.323 applications. The results in Figure 27, illustrate the different treatment received by the gold and silver quality real-time video traffic. Gold quality traffic is immuned from bursty cross traffic, while the silver quality video shows occasional drop at instances when bursts occur, though not drastic.

6 Conclusions and Future Work

Q-Pod opens the door for constructing realizable QoS architecture for existing enterprise networks, with a diverse user base executing legacy applications with heterogeneous QoS requirements running on disparate end systems with legacy operating systems. The key contributions of this paper are twofold. First, we describe in detail the performance mechanisms and algorithms in Q-Pod's platform independent design that allow low footprint, transparent and deployable QoS support for legacy applications on legacy operating systems. We present mechanisms to achieve dynamic session mapping of network sessions to their owning applications and their appropriate processing based on QoS requirements. This objective is enabled by the use of a kernel level network loadable driver, called Q-Driver, and a user-space control application called the Q-Manager. We present a novel high performance user-kernel communication scheme that avoids overheads by using a lock-free shared memory

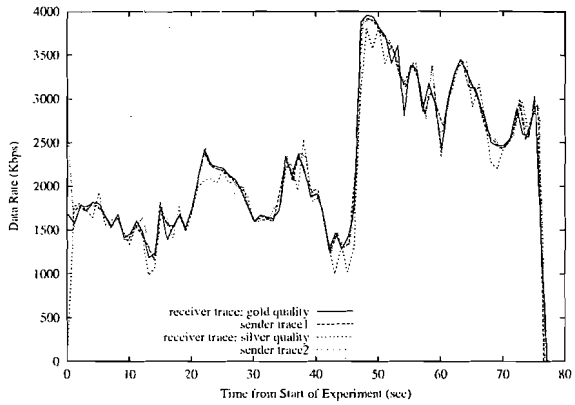

Figure 27: Real-time multimedia quality differentiation using destination address based admission control. The effect of bursty traffic is noticeable, though not to harsh, for the silver quality video stream.

which preserves data consistency under concurrent access between the user and kernel modules of Q-Pod. In addition it allow fast lookups and facility to maintain fine-grained traffic measurements. Based on prototype implementations of Q-Pod on Windows XP and Linux we demonstrate the performance and low footprint of Q-Pod. We present low level overhead measurements and evaluate Q-Pod performance under varying workloads and high duress. The results confirm the scalability of Q-Pod. Second, we present Q-Pod's functional features which, capitalizing on QoS mechanisms exported by legacy routers, enables scalable end-to-end QoS, integrating the user, applications, end systems, network core and service provider, in an enterprise environment, on a "turn-key" basis. We present end-to-end QoS benchmark for enterprise level applications—VoIP, grid computing and real-time multimedia content distribution. The benchmarks are performed on real network testbed consisting of 9 CISCO 7200 series routers. AF PHB exported by these routers is utilized. Our results show that scalable and deployable QoS with Q-Pod end system support is viable and can *enrich* the services attainable on legacy enterprise systems.

References

[1] Dassfnet: A c++ implementation of ssfnet. http://www.cs.dartmouth.edu/ghyan/dassfnet/overview.htm.

[2] Delivering predictable host integration services. http://www.cisco.com/warp/public/cc/so/neso/ibso/ibm/s390/phost_w

[3] International telecommunication union. http://www.itu.org.

[4] Netfilter. http://www.netfilter.org.

[5] OpenH323 Project. http://www.openh323.org.

[6] Tcpdump. http://www.tcpdump.org.

[7] W. Almesberger. Linux traffic control - next generation. In *9th International Linux System Technology Conference (Linux-Kongress 2002)*, pages 95–103, 2002.

[8] Christina Aurrecoechea, Andrew T. Campbell, and Linda Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, 1998.

[9] B. Bethala and K. Park. Personal communication, Network Systems Lab, Dept. of CS, Purdue U.

[10] S. Blake, D. Black, M. Carlson, Z. Wang E. Davies, and W. Weiss. *An architecture for differentiated services*. RFC 2475, Internet Engineering Task Force, 1998.

[11] J. Bruno, E. Gabber, B. Ozden, and A. Silberschatz. The eclipse operating system: Providing quality of service via reservation domains. In *USENIX '98*, pages 235–246.

[12] S. Chen and K. Park. A distributed protocol for multi-class QoS provision in noncooperative many-switch systems. In *Proceedings of IEEE International Conference on Network Protocols*, pages 98–107, 1998.

[13] D. Clark and W. Fang. Explicit allocation of best-effort packet delivery service. *IEEE/ACM Transactions on Networking*, 6(4):362–373, 1998.

[14] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual*, volume 3: System Programming Guide. http://developer.intel.com/design/pentium4/ manuals, 2003.

[15] Microsoft Corporation. *Microsft Platform SDK (Windows XP)*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/qos/qos/windows_xp_qos_support.asp.

[16] Microsoft Corporation. *Windows XP Driver Development Kit Help*. Windows DDK/Network Devices And Protocols/Design Guide/Intermediate NDIS Drivers, July 2001.

[17] R. L. Cruz. Quality of service guarantees in virtual circuit switched networks. *IEEE Journal on Selected Areas in Communication*, 13(6):1048–1056, 1995.

[18] G. de Veciana, G. Kesidis, and J. Walrand. Resource management in wide-area ATM networks using effective bandwidths. *IEEE Journal on Selected Areas in Communication*, 13(6):1081–1090, 1995.

[19] C. Dovrolis, D. Stiliadis, and P. Ramanathan. Proportional differentiated services: Delay differentiation and packet scheduling. In *Proceedings of ACM SIGCOMM '99*, 1999.

[20] A. Feldmann, A. Gilbert, W. Willinger, and T. Kurtz. The changing nature of network traffic: Scaling phenomena. *Computer Communication Review*, 28(2):5–29, 1998.

[21] R. Gopalakrishnan and G. Parkulkar. Efficient user space protocol implementations with qos guarantees using real-time upcalls. *IEEE/ACM Transactions on Networking*, 6(4):374–388, 1998.

[22] Pawan Goyal, Xingang Guo, and Harrick M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, 1996.

[23] J. Heinane, F. Baker, W. Weiss, and J. Wroclawski. *Assured Forwarding PHB group*. RFC 2597, Internet Engineering Task Force, 1999.

[24] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2:1–15, 1994.

[25] J. MacKie-Mason and H. Varian. Economic FAQs about the Internet. In L. McKnight and J. Bailey, editors, *Internet Economics*, pages 27–63. MIT Press, 1996.

[26] C. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems*, 1994.

[27] K. Nichols, S. Blake, F. Baker, and D. Black. *Definition of differentiated services fields (DS Field) in the IPv4 and IPv6 headers.* RFC 2474, Internet Engineering Task Force, 1998.

[28] K. Nichols, V. Jacobson, and L. Zhang. *A two-bit differentiated services architecture for the Internet.* RFC 2638, Internet Engineering Task Force, 1999.

[29] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, 1993.

[30] A. Parekh and R. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking*, 2(2):137–150, 1994.

[31] K. Park, G. Kim, and M. Crovella. On the relationship between file sizes, transport protocols, and self-similar network traffic. In *Proceedings of IEEE International Conference on Network Protocols*, pages 171–180, 1996.

[32] S. Radhakrishnan. Linux - Advanced networking overview. http://qos.ittc.ukans.edu/howto/howto.html.

[33] Carl A. Waldspurger and William E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, 1995.

[34] J. Wroclawski. *The use of RSVP with IETF Integrated Services.* RFC 2210, Internet Engineering Task Force, 1997.